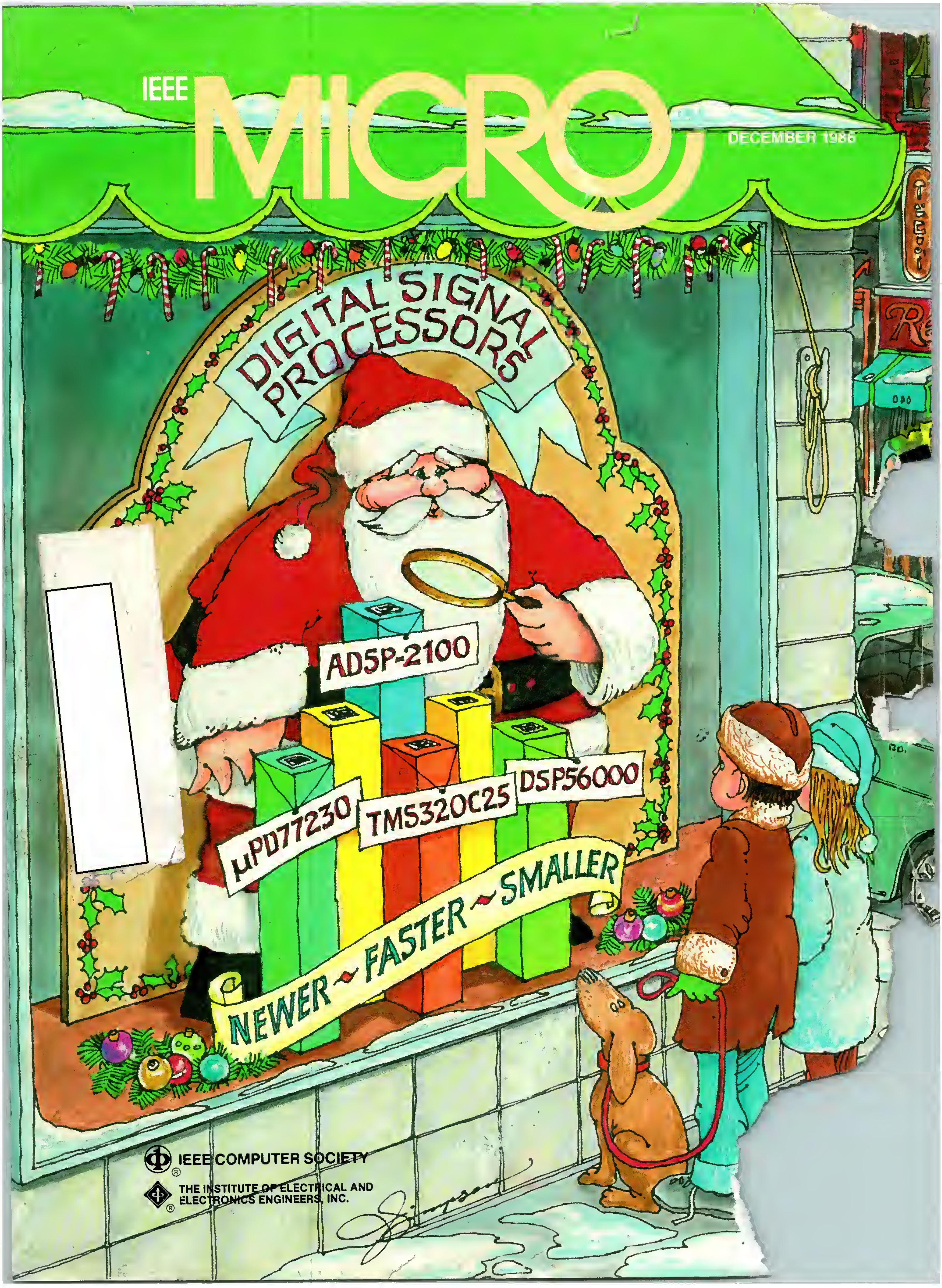
IEEE MICRO

DECEMBER 1986



IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND
ELECTRONICS ENGINEERS, INC.

IEEE MICRO

On the cover
With its original Jay Simpson cover celebrating both digital signal processing and the holiday season, *IEEE Micro* wishes all readers a very pleasant and prosperous new year.

Article design/production:
Alexander Torres and Miriam Wiegel
Typesetting: Fagen Graphics

In the next issue:
Operating Systems, Multiprocessing, and DSP
See p. 88.

Volume 6 Number 6 (ISSN 0272-1732) December 1986

Feature Articles

Digital Signal Processing

Departments

IEEE MICRO

Editor-in-Chief: James J. Farrell III,
VLSI Technology Incorporated*

Associate Editor-in-Chief:
Joe Hootman, University of North Dakota

Editor and Publisher: True Seaborn
Assistant Publisher: Gary Carter
Managing Editor: Marie English
Contributing Editor: Joe Schallan
Assistant to the Publisher: Patricia Paulsen
Advertising Coordinator: Carole L. Porter
Membership/Circulation Manager: Christina Champion
Art Director: Jay Simpson
Production Supervisor: David Gaines

*Submit six copies of all articles and special-issue proposals to James J. Farrell III, 10220 South 51st Street, Phoenix, AZ 85044; (602) 893-8574.

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the Computer Society of the IEEE: IEEE Headquarters, 345 East 47th St., New York, NY 10017; IEEE Computer Society West Coast Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Annual subscription: $12.00 in addition to IEEE Computer Society or any other IEEE society member dues. Nonmember prices: available on request. Single-copy prices: members $7.50; nonmembers $15.00. This journal is also available in microfiche form.

Undelivered copies: Send to 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

Postmaster: Send address changes to *IEEE Micro*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. Second class postage is paid at New York, NY, and at additional mailing offices.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services in New Products, Product Summary, MicroReview, MicroNews, and MicroLaw, reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the Computer Society of the IEEE. Send editorial correspondence to *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All submissions are subject to editing for style, clarity, and space considerations. *IEEE Micro* subscribes to The Computer Press Association's code of professional ethics.

BPA

From the Editor-in-Chief

1986 Report

Because of the strict budgetary measures instituted a year ago, *IEEE Micro* should show a modest profit in 1986. Even so, this was a pretty tough year for most trade magazines. During this ongoing malaise in the computer and semiconductor industry we lost about 1500 readers. We were not hurt by industry cutbacks in advertising because we simply did not have much to lose.

To ensure future health of the magazine, increases in our page count, and—as many readers are requesting—a move toward a monthly magazine, we need to increase our readership and attract more advertising. I do not foresee *IEEE Micro* becoming awash in ads, as some magazines are, but ads do have value to the reader as well as being revenue sources for the magazine. To help expose *IEEE Micro* to a wider potential audience and thereby increase readership, we will be offering "special" introductory subscriptions, forming alliances with conferences, and actively pursuing subscribers wherever we can. In the process of becoming more visible, we should become more attractive to advertisers. I need your help. If you have a suggestion to offer, or know of a group or event that may be a candidate for our new subscriber campaign, please let me know. If you or your employer are considering an advertisement, please give me a call, and I will put you in touch with a representative.

The "special theme" issues of 1986 have been a resounding success, judging from the mail and telephone messages I have been receiving. This issue on digital signal processing edited by Bob Morris closes out our 1986 editorial calendar. I would be remiss if I did not publically thank all of our 1986 guest editors for a splendid job:

June 1986, Arthur R. Miller and Rhonda Alexis Dirvin (networking),
August 1986, Victor K. L. Huang and Priscilla M. Lu (operating systems),
October 1986, Barry W. Johnson (multiprocessing), and
December 1986, L. Robert Morris.

We have a very healthy backlog of excellent articles in addition to the 1987 special theme issues. We shall reduce this queue by publishing four or five articles in the February 1987 issue. TRON comes in April.

I must admit that I am becoming very excited about the TRON issue. During the past two weeks major news articles concerning TRON have appeared not only in *Electrical Engineering Times* and *Electronic News* but also in the *Wall Street Journal*. Ken Sakamura, of your *IEEE Micro* editorial board, was named as the prime device and system architect in all three publications. TRON stands for The Realtime Operating System Nucleus and will be thoroughly covered by Sakamura in April.

The mailbag was a little light this time, containing only 20 comment cards. As usual, I summarize the comments:

"I would like to subscribe," P.A., Netanya, Israel (A good start—J.F.)
"I like tutorial-type articles, even if they are blatantly motivated by commercial interest." A.B., Manchester, NH
"...would like to see more digital signal analysis." E.A., McLean, VA
"...Benchmark article was good but (omitted) operating systems-type operations." K.S., Lexington, MA
"I liked VRTX...more academic authors." T.F., Keene, NH
"Another good issue...I don't feel lack of ads is very bad, although I'm sure it would be nice for revenue, though!" B.H., Albuquerque, NM
"All articles should start on right-hand page to simplify rip-and-save." Anon. (Ouch! Would you consider saving the whole magazine?— J.F.)
"I liked everything. Just go on! More comparisons of chips and architectures." P.L., Rio de Janeiro, Brazil
"VRTX was fascinating." S.S., Falls Church, VA
"I liked the looks of all the articles... *IEEE Micro* is on the top of the pile." C.T., Cropwell, AL
"Overall, (this was) an excellent issue (networking)." J.B., Middletown Springs, VT

Final notes: Henriecus Koeman has completed his terms on your editorial board and leaves this month. In addition to his excellent editorial reviewing work, Henriecus provided valuable ideas in our refereeing process, which I have since implemented. Thank you, Henriecus.

To those readers observing Christmas or Chanukah, we wish a joyous holiday season. To all of our readers, we wish a New Year filled with health, peace, prosperity, and the benefits of technology.

Jim Farrell

Jim Farrell

Letters to the Editor

Benchmark article stirs reader response

To the Editor:

With regard to benchmarks—yes, yes, YES. There are few benchmarks with any meaning that get published; that is, most of them are in somebody's advertising literature and are, shall I say, suspect. I found the article in the August issue of *IEEE Micro* very interesting.

One slight problem with that article was the final summary of results; the authors took an arithmetic mean when they should have taken a geometric mean. Their main conclusions are the same, but the advantage of the 68020 over the other processors is not quite as great as it appears from their numbers. I have enclosed the correct version of their Table 3, along with an article from the *CACM* showing why this is the correct method. (Editor's note: The *CACM* article is not reprinted here because of its length; see *Comm. ACM,* Vol. 29, No. 3, Mar. 1986, pp. 218-221.)

Bruce Walker
San Pedro, CA

Table 3.
Revised using geometric means.

	Dynamic memory		Static memory	
Processor	Mean	Ratio	Mean	Ratio
80286	9.91	3.55	5.40	2.78
80386	4.91	1.76	2.44	1.26
68000	12.87	4.61	8.39	4.32
68020N	5.41	1.94	2.67	1.38
68020C	2.79	1.00	1.94	1.00
32032	10.26	3.68	8.37	4.31
32100N	9.00	3.23	4.73	2.44
32100C	4.18	1.50	3.22	1.66

Thayne Cooper replies

We wish to thank all of you who read our 32-bit microprocessor benchmark article. We hope the information was useful as well as interesting to many of you.

The letter by Bruce Walker addresses an issue which is always of interest in an article like ours: How should the information be presented? We read the article referenced by Walker with interest and suggest that others read it also.

We do note, however, that the article referenced by Walker treats the use of the geometric means on normalized numbers of individual tests. The summary we provided did not use normalized test numbers. Rather, it took the arithmetic means of the raw unnormalized numbers and then normalized those means.

We answered the question of information presentation by supplying the raw data of the tests along with a summary. This provides the opportunity for other kinds of summaries to be made. Walker has done that and arrived at essentially the same results.

To the Editor:

While we at National Semiconductor Corporation were pleased that one of the members of our Series 32000 microprocessor family was included in the article, "A Benchmark Comparison of 32-bit Microprocessors" (*IEEE Micro,* Aug. 1986, p. 53), we were disappointed to find that the microprocessor tested was the 10-MHz NS32032 and not the newest member of the family, the 15-MHz NS32332.

This MPU, which is both fully upward and downward software compatible with the other members of the Series 32000 family and has such speed-increasing features as an improved microinstruction set, larger instruction prefetch queue, and burst access mode, has been available as a 10-MHz part since October of 1985, with the 15-MHz version available since March of 1986.

Our own in-house executions of the *EDN* benchmarks, run on National Semiconductor's NS32032-10-based DB32000 demonstration board, have produced slightly faster results than were shown in the article, due to our use of zero-wait-state RAM. Execution of the same benchmarks on our DB332 demonstration board, utilizing the 15-MHz NS32332 and zero-wait-state RAM, have shown an overall speed increase, as determined from the mean of the test times, of approximately 2.2 times.

National Semiconductor is working with the authors of the article to ensure that the NS32332-15 is included in their next round of benchmark tests and looks forward to results which will clearly show the architectural improvements of the NS32332.

David Raulino
Santa Clara, CA

Cooper's response

Since our report in the August issue of *IEEE Micro* on 32-bit microprocessor performance, our evaluations have continued. The National 32332 (follow-on to the 32032) has been received and the same benchmarks run on it. The following are the numbers which can be added to the tables found in the August article:

Table 1.

32332	(15)	8.45	8.73	4.34	6.43	7.41

Table 2.

32332	(15)	6.92	6.26	2.42	3.79	6.15

Table 3.

32332	(15)	7.07	2.49	5.11	2.57

Bus wars

To the Editor:

A word of thanks to Bob Stewart for his very kind reference to me in his article in the August *IEEE Micro*. Also, Mildred and I were very amused at Bob's ". . .Bus Wars" cartoons—this is an accomplishment we'd never suspected.

We still have very pleasant memories of the Oregon expedition, and especially of that final Saturday we spent together. We hope all is well with Bob and that there'll be another opportunity to get together before we're too old and grey.

Matthew Taub

To the Editor:

The so-called "history" of microcomputer bus standards development presented in cartoon form in the August issue (*IEEE Micro*, "MicroStandards: Promises, promises, promises," p. 66) grossly misrepresents at least those events with which I am personally familiar. This is particularly egregious in that I know author Stewart to be well aware of the facts and the correct information was readily available to your editors (see, e.g., the P896 PAR, *IEEE Micro,* Vol. 1, No. 1., p. 67; or Wescon/81 Professional Program Session Record 27/5).

To cite a few specific examples:

Cash Olsen had absolutely nothing to do with the formation of the P896 working group. Olsen was the chairman of a subgroup (the Futurebus subcommittee) formed within the P696 project in mid-1978. This subgroup, of which I was a member, was established to identify future microprocessor bus requirements in general, and not at Olsen's behest (panel 10). In June 1979, frustrated by the lack of progress within this subcommittee, I organized the study group on advanced microcomputer system buses that prepared the project authorization request resulting in P896. The P896 activity never met in Berkeley (panel 14). Note also that the misleading use of the term Futurebus for the P896 activity began at least three years after its inception.

The recognition of Taub's development of the arbitration mechanism (panel 11) was, in fact, quite belated. Gustavson presented the concept to P896 without attribution and long before it became known that it had originated with Taub. It is my impression that it was the publicity surrounding its use in P896 that drew attention to the fact that it had actually originated with Taub.

The serial intermodule communication concept—one of the things most violently objected to by opponents of the early P896 drafts, but which has since been incorporated into both the VMEbus and Multibus II—originated with Rollie Linser, and the attribution to me in panel 15 is quite simply false, as is the one in panel 17. Stewart did not attend the Boulder meeting, but I specifically advised him of Linser's contribution when he telephoned me early this year seeking background material for the original presentation of the material in question. There was, of course, ample opportunity to verify the other information at that time. Jean-Daniel Nicoud of EPF-Lausanne and the European participants whose activities he coordinated until 1982 also made many valuable contributions, including the low-cost, single-connector approach since adopted by Nubus.

The end of Nicoud's and my active participation in the P896 effort (panel 19) is also misrepresented. In December 1981 (not 1980 as shown), the P896 working group voted—16 in favor and five (three of whom failed to meet the requirement to state the modifications necessary to cause them to change their vote) opposed—to take Draft 4.1.1 to the MSC with a request to approve distribution for public comment. Contrary to the impression given in Stewart's comic strip, the rejection of this request by the other members of the MSC at its January 14, 1982, meeting was not unanimous, and Nicoud was not even present.

As was made clear by my statement at the time, a copy of which was provided to the then chairman of the MSC but misrepresented in the minutes of the meeting, my resignation was brought about not by the denial of the working group's request, but by the committee's general failure to prevent the holders of minority viewpoints from indefinitely delaying progress. The result, as I predicted then, has been a failure to achieve results commensurate with the enormous amount of effort that has been devoted by working groups over the past nine years. In the case of P896, for example, the MSC has manifestly failed in the objective stated in the PAR, namely to provide an alternative to the development of yet another generation of incompatible de facto bus standards, namely, those previously mentioned plus VAXBI.

Judging by the quality of the reporting of events with which I am familiar, the material in question may possibly belong in the funny pages, but clearly has no place in a publication of the IEEE. Kindly consider this a request for a public retraction.

Andrew Allison
Los Altos Hills, CA

Response from Robert G. Stewart

Ko Ko: Your Majesty, it's like this: It is true that I stated that I had killed Nanki-Poo—
Mikado: Yes, with most affecting particulars.
Pooh Bah: Merely corroborative detail intended to give artistic verisimilitude to a bald and. . .

The Mikado,
W. S. Gilbert and Arthur Sullivan

Andrew Allison devoted hard and constructive efforts for years to the standardization activities of the IEEE Computer Society. He was the first MicroStandards editor of *IEEE Micro.* I sincerely apologize if he felt affronted by the material in my August column, "A Historical? View of IEEE Standards During the Great Bus Wars."

His letter brings up several aspects of the 896 Futurebus project, some of which I agree with, and some which I don't. Cash Olsen was dubious of the value of standardizing the S-100 MITS Altair bus (now IEEE 696). The talk of Tony Pietsch to the Microprocessor Standards Committee emphasized the need for more advanced buses. That led to the initiation of a study group with Cash as chair of the study group acting under the 696 PAR to look at other possible bus efforts. He looked at a "Home Bus" that never went very far

continued on page 82

Guest Editor's Introduction

Digital Signal Processing Microprocessors: Forward to the Past?

L. Robert Morris
Carleton University and DSPS Inc., Ottawa

In 1986, a number of new digital signal processing microchips were announced by semiconductor manufacturers whose names are familiar to *IEEE Micro*'s readers: Analog Devices, Motorola, National Semiconductor, NEC, Texas Instruments, and Philips/ Signetics. What are DSP micros? What makes them different from the general-purpose micros offered by many of the same companies? What are their applications? Why should *IEEE Micro*'s readers be interested in such devices? This brief introduction and the articles which follow will attempt to answer most of these questions.

DSP micros share one feature: speed in the execution of certain algorithms. As was first noted in a 1983 survey,[1] these processors are, in effect, reduced-instruction-set computers optimized for the fastest possible execution of addition, subtraction, multiplication, and shifting instructions. In early DSP micros especially, a reduced instruction set, which can be implemented in a small area of silicon, is accompanied by single-cycle multiplication and shifting, which are accomplished by devoting a relatively large area of silicon to an array multiplier and a barrel (or combinatorial) shifter. In contrast, most current general-purpose micros still effect such operations via multiple-cycle, microcoded instructions that make use of the arithmetic unit's single-cycle, parallel-add and single-bit shift capability. Since integer multiplication and shifting are statistically unimportant for most programs that run on general-purpose micros, designers of such devices prefer to devote large areas of silicon to implementation of larger, more versatile instruction sets (sometimes including floating-point in the on-chip microcode), memory management, or cache memories.

The implication of the above—that fast integer multiplication and shifting are considered crucial to digital signal processing software—is correct. In fact, the software implementation of the most common digital signal processing algorithm, an n-tap finite-length impulse response (FIR) filter, essentially consists of n multiply/accumulates. These instructions are executed once for every signal sample that is input (at rates typically of 8 kHz and above). Most of the newer DSP micros can accomplish each multiply/accumulate in a single cycle of about 100 ns! This is one to three orders of magnitude faster than most general-purpose micros. For example, a 16-MHz 80386—a state-of-the-art micro which effects register-to-register 16-bit addition (ADD) in only 125 ns—requires about 1250 ns for a 16 × 16-bit multiplication (IMUL), and a 5-MHz 8088 requires 32,000 ns for the same instruction! Other important DSP algorithms—the fast Fourier transform, or FFT, for example—require many more addition/subtractions than multiplications, but even for these algorithms the relatively slow multiply on general-purpose processors represents a significant bottleneck.

The first DSP micro, the Intel 2920, appeared nearly a decade ago. It was followed by the AMD 2811, the NEC μPD7720, and, in 1982, the Texas Instruments TMS32010. While the 2811 and 7720 both had on-chip array multipliers, both were ROM-programmable only and had relatively small data and program address spaces. The 32010 was the first DSP micro that could execute instructions at full speed from an off-chip program RAM, and it could also accommodate a program nearly an order of magnitude larger than the 7720 could.

The articles in this issue will reveal both similarities and differences between DSP and general-purpose micros. For example, DSP micros employ many speed- and efficiency-related design strategies also employed in regular micros: pipelining of instructions, use of addressing modes that efficiently access relevant data structures (e.g., autoincrement and autodecrement modes for arrays and an indexed ad-

0272-1732/86/1200-0006$01.00 © 1986 IEEE

Table 1.
Complex FFT performance (times in ms).

	Fixed-point				Floating-point
	Number of points				Number of points
	64	256	1024		1024
1976 DSP				1976 DSP	
SPS 61	0.21	1.10	4.89	AP-120B	4.75
SPS 21	1.08	2.52	8.13	MAP-100	60.00
1976 general-purpose				1976 general-purpose	
PDP-11/55	4.22	20.00	118.00	PDP-11/55 with FP-11C	168.00
1986 DSP				1986 DSP	
DSP56000 (20 MHz)	0.140	0.71	4.99	μPD77230 (15 MHz)	12.50
TMS320C25 (40 MHz)	0.217	1.22	7.10		
ADSP2100 (32 MHz)	0.319	1.52	7.19		
TMS32020 (20 MHz)	0.434	2.44	14.18		
LM32900 (20 MHz)	0.550		13.40		
5010/11 (8 MHz)		3.30	33.00		
TMS32010 (20 MHz)	0.535	6.30	30.00		
1986 general-purpose				1986 general-purpose	
80386 (16 MHz)	1.667	9.50	50.00	80386/387 (16 MHz)	100.00 (est.)
80286 (8 MHz)	3.800	20.00	110.00		
8086 (8 MHz)	11.000	55.00	310.00		
8088 (5 MHz)	19.000	110.00	610.00	8088/87 (5 MHz)	995.00

Benchmark sources:

- DSP56000, μPD77230, ADSP2100—*IEEE Micro*, Dec. 1986,
- LM32900, 5010—information from manufacturers,
- PDP-11/55, TMS32010, 80386, 80286, 8088, 8086, 8087, 80387—DSPS Inc.,
- TMS32020—TI's *Details on Signal Processing*, Nov. 1985, and
- TMS320C25—extrapolated from TMS32020.

Smaller FFTs are often proportionally more efficient than larger ones because

- in-line code can often be used for $n = 64$ points or less, and
- smaller FFTs contain a larger fraction of more efficient, faster, zero-multiply butterflies.

The FFT is a representative and realistic DSP benchmark since it contains a mixture of multiplications and additions and its computational kernels (the butterflies) are of nontrivial program complexity compared to the single instruction required for FIR filters on many DSP micros.

dressing mode for FFTs), and use of "clean" subroutine calling and address passing protocols. Differences include DSP micros' use of the dual-bus Harvard architecture, which enables simultaneous fetching of instructions and data; special DSP-related addressing modes (e.g., index computation modulo an arbitrary number, automatic circular queue or free data move for FIR filters, and bit reversal for FFTs); extra addressing ALUs; and special interfaces to serve specific fields of application (e.g., serial interfaces for codecs in telecommunications).

How were DSP algorithms implemented in the pre-DSP micro era? In the early 1970's, array processors—first fixed-point and then floating-point—were available for real-time execution of many audio-bandwidth DSP algorithms. These machines varied in cost from $10,000 to $50,000 and typically consisted of a rack-mounted unit weighing upwards of 100 lbs. and consuming about a kilowatt of power. These attributes generally limited the use of array processors to large laboratories and certainly precluded the inclusion of such machines as subcomponents in OEM systems. The data in Table 1 reveals an interesting fact: the 1976 array processors and the 1986 DSP micros have comparable FFT execution times, and the same holds for 1976 general-purpose minicomputers and 1986 general-purpose micros! *Thus, today's DSP and general-purpose micros exhibit approximately the same performance as their decade-old ancestors.* Further, comparisons of their architectures do not reveal that any startling changes have occurred since 1976.

What *has* occurred, of course, is a three-order-of-magnitude reduction in cost, size, weight, and power consumption. It is the combination of 1976 array processor performance with 1986 microchip attributes that has both quantitatively and qualitatively changed the extent to which theory can be applied to the practical solution of problems in signal processing, communications, and control, and in new disciplines such as artificial intelligence. In many cases, the computer simulation traditionally carried out as a precursor to system realization via hardwired logic can now become the cost-effective implementation via software on a DSP micro.

DSP micros are now on the verge of surpassing their array processor ancestors in architectural complexity and sophistication as well as in performance. Thus, the theme finally becomes "Forward to the Future." VLSI allows active device densities and signal propagation times not possible a decade ago. And, fortunately, semiconductor technologies have not yet hit a "brick wall" in terms of speed. Gallium arsenide (GaAs) transistors and high-electron-mobility transistors (HEMTs) in particular suggest that another "easy" order-of-magnitude improvement in performance is not unreasonable to anticipate, even with existing architectures. Although DSP devices having parallel and dataflow architectures have appeared, at present they have not achieved the user acceptance of more conventional "sequential" processors. This is partially due to the fact that the present DSP micro user anticipates that performance enhancements requiring neither changes to algorithms nor even changes to software will continue to appear due to clock-speed-related semiconductor progress alone!

We should discuss one other possible scenario.[1] Note that the fastest general-purpose micros already approach the performance of the slowest DSP micros: a 16-MHz 80386 computes a 1K, complex, fixed-point FFT only 66 percent slower than a 20-MHz TMS32010 (see Table 1 again). With the newest versions of general-purpose micros already incorporating a DSP-like dual-bus architecture (for example, the Motorola 68030[2]), the obvious next step—integration of an array multiplier and a barrel shifter into general-purpose micros—cannot be far off. Since these two devices make possible fast floating-point multiplication and addition, respectively, and since floating-point performance "sells," most semiconductor manufacturers are on the verge of taking this step.

Although the resulting general-purpose micros will still lack some special instructions and architectural attributes that help in achieving maximum DSP performance, it is entirely conceivable—with GaAs technology already commercially viable in 1986[3,4]—that by incorporating GaAs/HEMT transistors they can achieve a performance of 100 MIPS and upwards and make special-purpose DSP micros unnecessary in many DSP applications.

References

1. L. R. Morris, "Real-Time Digital Signal Processing with Commentary on the Texas Instruments TMS32010 and the DEC J-11," *Trends & Perspectives in Signal Processing*, Vol. 3, No. 1, Mar. 1983, pp. 9-14.
2. "First Look at Motorola's Latest 32-bit Processor," *Electronics*, Vol. 59, No. 31, Sept. 18, 1986, pp. 71-75.
3. B. C. Cole, "GaAs LSI Goes Commercial," *Electronics*, Vol. 59, No. 31, Sept. 18, 1986, pp. 57-60.
4. "Bit-Slice ICs Kick Off Era of Commercial GaAs LSI," *Electronics*, Vol. 59, No. 31, Sept. 18, 1986, pp. 61-64.

L. Robert Morris is a professor of systems and computer engineering at Carleton University in Ottawa and is also president of DSPS Digital Signal Processing Software, Inc. His areas of interest include DSP algorithm/DSP micro architecture interactions, signal processing, and time/space optimization of programs for DSP micros. He is currently involved in a multiuniversity Canadian project for developing a DSP-micro-based, multichannel cochlear implant. He obtained a BASc in electrical engineering from the University of Toronto and a PhD in speech communications from the University of London (Imperial College), England.

Morris can be reached at Systems Engineering, MacKenzie Bldg., Room 377, Carleton University, Ottawa, Ont. K1S 5B6, Canada.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 195 Medium 196 Low 197

Announcing new PC SIMSCRIPT II.5 . . . with animation

SIMSCRIPT II.5 with animation now on personal computers

free trial -- see how SIMSCRIPT II.5 helps you build a realistic model

the complete SIMSCRIPT II.5 on a PC

SIMSCRIPT II.5 for personal computers is the same popular simulation language that is now widely used on mainframes.

You can now build realistic models of military, manufacturing, communications, logistics, transportation or other systems on your PC.

PC SIMSCRIPT includes a new programming environment that makes it easy for you to develop, verify, modify, and enhance simulation models on a personal computer.

natural method of modelling

When building a model in SIMSCRIPT, you describe the simulated system as consisting of certain types of entities: perhaps workers, machines and jobs in a simulated factory; or flights and airports in a simulated air transport system; or jobs, processors, channels, and I/O devices in a simulated computer system.

For each type of entity you give names to the attributes that characterize it.

You also name the sets an entity type may belong to, and the sets it may own.

Since your model is English-like, with names that you choose, it reads like a description of the simulated system. The model can be read and verified by non-programmers who understand the system under study.

This makes your model development, validation and evolutionary changes much easier.

large models on your PC

Your model and data are not limited by the size of the PC. SIMSCRIPT is the only simulation tool that automatically makes use of the hard disk as a memory extension.

reduced cost

SIMSCRIPT II.5® is a well established, standardized, and widely used language with proven software support.

Experience has shown that SIMSCRIPT II.5 reduces simulation programming time and cost severalfold compared to other simulation techniques.

animated and graphical results

With PC SIMSCRIPT II.5® you build models that can show an animated picture of the system under study. Observing the simulation improves understanding of the system and builds confidence in the model.

Because you see the operation of the simulated system and can easily try alternatives, the time and cost of system analysis are sharply reduced.

computers with SIMSCRIPT II.5

1. IBM Personal Computer AT, XT, PC or compatible, with a hard disk.

2. Most Mainframe computer types including IBM, CDC, VAX, Univac, Prime, Gould, Data General and Honeywell.

SIMSCRIPT II.5 and PC SIMSCRIPT II.5 are registered trademarks and service marks of CACI, INC.-FEDERAL

free trial

The free trial package contains everything you need to try SIMSCRIPT II.5 on your own computer.

We send you PC or Mainframe SIMSCRIPT II.5, installation instructions, sample models, and a complete set of documentation. You can build your own model or modify one of ours. No cost or obligation.

special offer free training

For a limited time we will also include free training. **Space is limited so act now to avoid disappointment.**

Call Rick Crawford at (619) 457-9681 to reserve your place.

free trial-learn the reasons for the broad and growing popularity of SIMSCRIPT II.5—no cost or obligation

special offer-return the coupon today and we will include one free course enrollment worth $850

Name ______________________________

Organization ______________________________

Address ______________________________

City __________ State __________ Zip __________

Telephone ______________________________

Computer __________ Operating System __________

Return to: IEEE MICRO

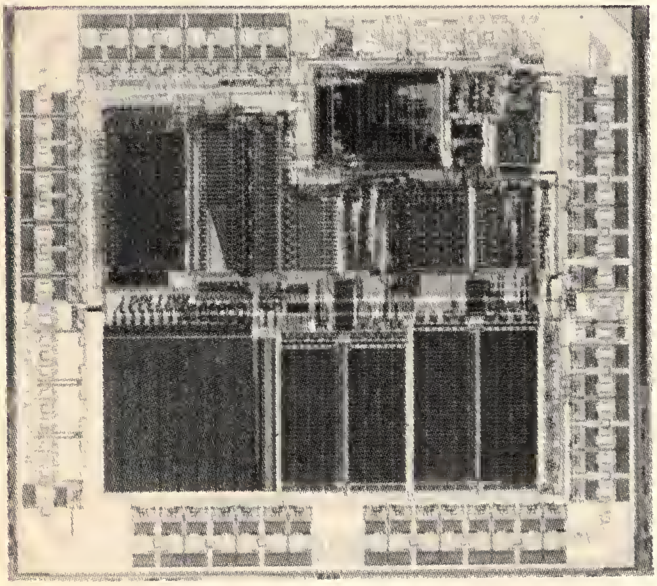
CACI

3344 North Torrey Pines Court
La Jolla, California 92037

Or, better yet,
call Rick Crawford at (619) 457-9681

The Texas Instruments TMS320C25 Digital Signal Microcomputer

Gene A. Frantz, Kun-Shan Lin,
Jay B. Reimer, and Jon Bradley
Texas Instruments Incorporated



Capable of 10 million operations per second, the newest member of the TMS320 family can serve as an inexpensive alternative to bit-slice processors or custom ICs in digital signal processing applications.

Digital signal processing encompasses a variety of applications, including digital filtering, speech vocoding, image processing, fast Fourier transforms, and digital audio.[1-5] All DSP applications have several characteristics in common. First, they employ algorithms that are mathematically intensive. An example is the finite-duration impulse response, or FIR, filter, which in the time domain takes the form

$$y(n) = \sum_{i=1}^{N} a(i) \cdot x(n-i), \qquad (1)$$

where $y(n)$ is the output sample at time n, $a(i)$ is the ith coefficient or weighting factor, and $x(n-i)$ is the $(n-i)$th input sample. From this equation, we can see that the FIR filter contains an abundance of multiplications and additions (that is, sums of products). This equation is the general form of an FIR filter[6] as well as the convolution of two sequences of numbers $a(i)$ and $x(i)$.[7] Both operations are fundamental to digital signal processing.

Second, DSP algorithms must be performed in real time; i.e., they must not produce a delay noticeable to the user. In a speech recognition system, for example, the algorithms must not produce a noticeable delay between a word being spoken and that word being recognized. In an image processing system, processing needs to be completed within a frame update period.

Third, all DSP applications involve the sampling of a signal. Referring to Equation 1, we can see that the output $y(n)$ is calculated to be the weighted sum of the previous N inputs. In other words, the input signal is sampled at periodic intervals, and the samples are multiplied by a weighting factor $a(i)$ and then added together to give the output result $y(n)$. In a typical DSP application, the processor must be able to perform arithmetic computations and effectively handle sampled data in large quantities.

Last, DSP systems must be flexible enough to incorporate improvements in the state of the art. Many DSP techniques are still developing, and therefore their algorithms tend to change. Speech recognition, for example, is presently an inexact technique still undergoing algorithmic modification. This implies that DSP systems need to be programmable so that they can easily accommodate revised algorithms.

Over the past several decades, digital signal processing machines have taken several forms in response to application need and available technology. Array processors have long been the accepted solution for the research laboratory and have been extended to end applications in some instances. However, as integrated circuit technology has matured, digital signal processing has migrated from the array processor to the bit-slice processor to the single-chip processor. This has brought the cost of DSP solutions down to a point that allows pervasive use of the technology.

The members of the TMS320 family of devices are examples of the single-chip digital signal processor. The first member of the family, the TMS32010, was introduced to the market in 1983.[8,9] It can perform five million DSP

0272-1732/86/1200-0010$01.00 © 1986 IEEE

operations per second, including the add and multiply functions[10] required in Equation 1. The newest member of the family, the TMS320C25, can perform 10 million DSP operations per second,[11] and it combines the multiply/accumulate functions into one single-cycle operation.

Basic TMS320 architecture

The fundamental attribute of a digital signal processor is fast arithmetic operations. The members of the TMS320 family,[10-12] like many other digital signal processors, achieve fast arithmetic operations by employing

- a Harvard architecture,
- a dedicated hardware multiplier,
- special DSP instructions, and
- extensive pipelining.

Use of these concepts allows a digital signal processor to handle a vast amount of data and execute most DSP operations in a one-cycle instruction.

The TMS320 family utilizes a modified Harvard architecture for speed and flexibility. In a strict Harvard architecture,[13,14] the program memory and data memory lie in two separate spaces, permitting a full overlap of the instruction fetch and execution. The TMS320 family's modification of the Harvard architecture allows transfers between the program space and data space, thereby increasing the flexibility of the devices in the family. This architectural modification eliminates the need for a separate coefficient ROM and also maximizes processing power by maintaining two separate bus structures (program and data) for full-speed execution.

The TMS320 family's dedicated hardware multiplier employs a 16 × 16-bit organization, which yields a 32-bit result and allows multiplication to take place in a single cycle. The special DSP instructions include DMOV (data move) and RPT (repeat), which speed up DSP operations. The extensive pipelining ensures maximum throughput for real-time applications.

The TMS320C25 architecture

The TMS320C25 digital signal processor is a microcomputer with a 32-bit internal Harvard architecture and a 16-bit external interface. It is a pin-compatible CMOS version of the TMS32020 microprocessor but has an instruction execution rate twice as fast and includes additional hardware and software features. The TMS320C25's instruction set is a superset of that of the TMS32010 and that of the TMS32020, and it maintains source-code compatibility with them. In addition, it is completely object-code-compatible with the TMS32020 so that TMS32020 programs can run unmodified on the TMS320C25. Some of the major features of the TMS320C25 are

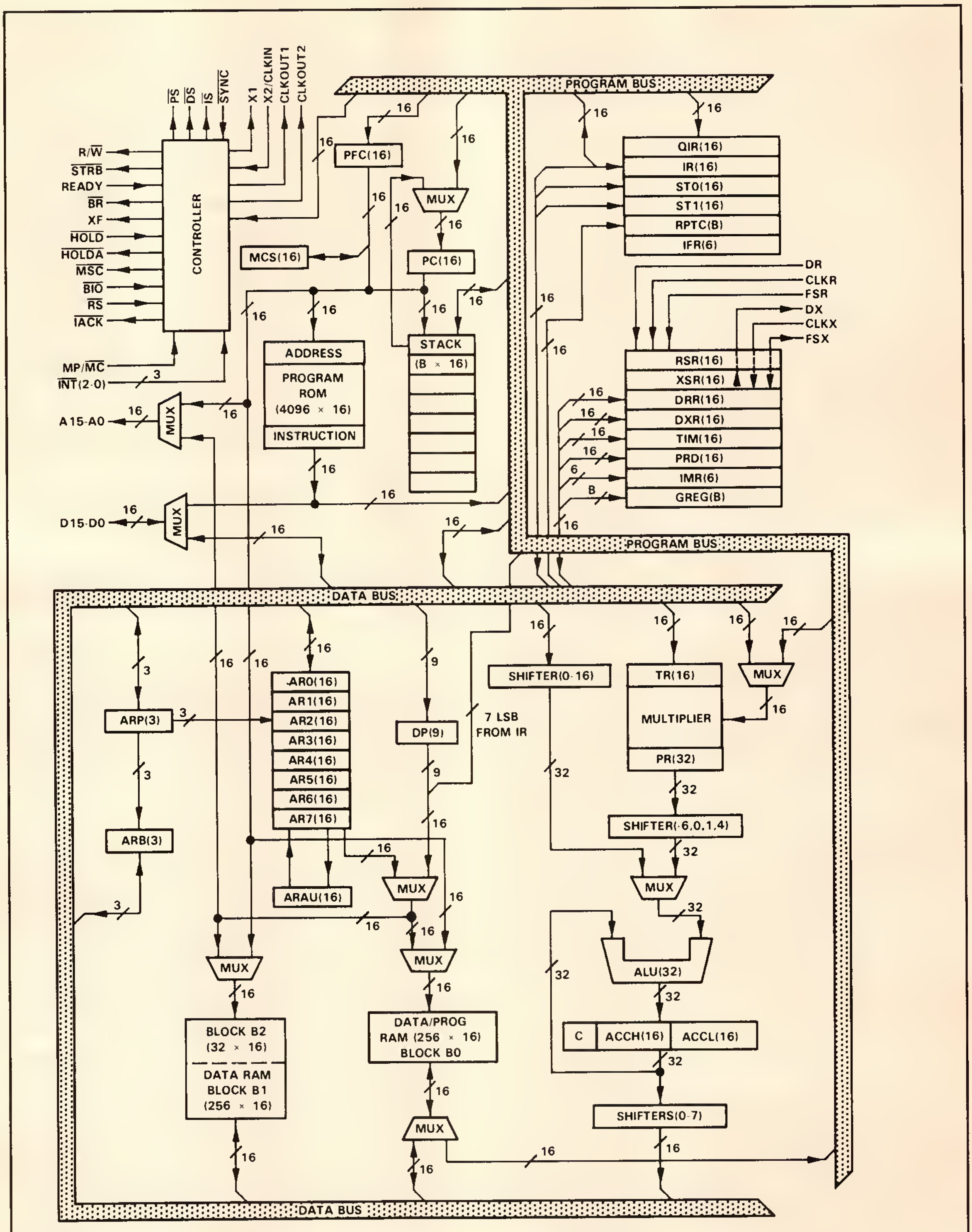
- a 32-bit ALU and accumulator,
- an instruction cycle time of 100 ns,
- a single-cycle multiply/accumulate,
- use of low-power CMOS technology with a power-down mode,
- 4K 16-bit words of on-chip masked ROM,
- 544 words of on-chip data RAM,
- 128K words of data/program memory space,
- eight auxiliary registers with a dedicated arithmetic unit,
- an eight-level hardware stack,
- a fully static double-buffered serial port,
- concurrent DMA that uses an extended hold operation,
- bit-reversed addressing modes for fast Fourier transforms,
- extended-precision arithmetic and adaptive filtering support,
- full-speed operation of data move instructions from external memory,
- an accumulator carry bit and related instructions, and
- fabrication in 1.8-μm CMOS and packaging in a 68-pin PLCC.

The 100-ns instruction cycle time provides a significant throughput advantage for many applications. Since most of the TMS320C25's instructions can execute in a single cycle, it can execute 10 million instructions per second. Most of the other features listed above also contribute to the TMS320C25's high throughput.

The TMS320C25 includes instructions to perform the data transfers between program space and memory space discussed earlier. Externally, the program and data memory spaces are multiplexed over the same bus so as to maximize the address range for both spaces and minimize the pin count of the device. Internally, the TMS320C25 architecture maximizes processing power by maintaining two separate bus structures, program and data, for full-speed execution.

Program execution in the device takes the form of a three-level instruction fetch-decode-execute pipeline. This pipeline is invisible to the user except in cases in which it must be broken, such as for branch instructions. In this case, the instruction timing takes into account the fact that the pipeline must be emptied and refilled.

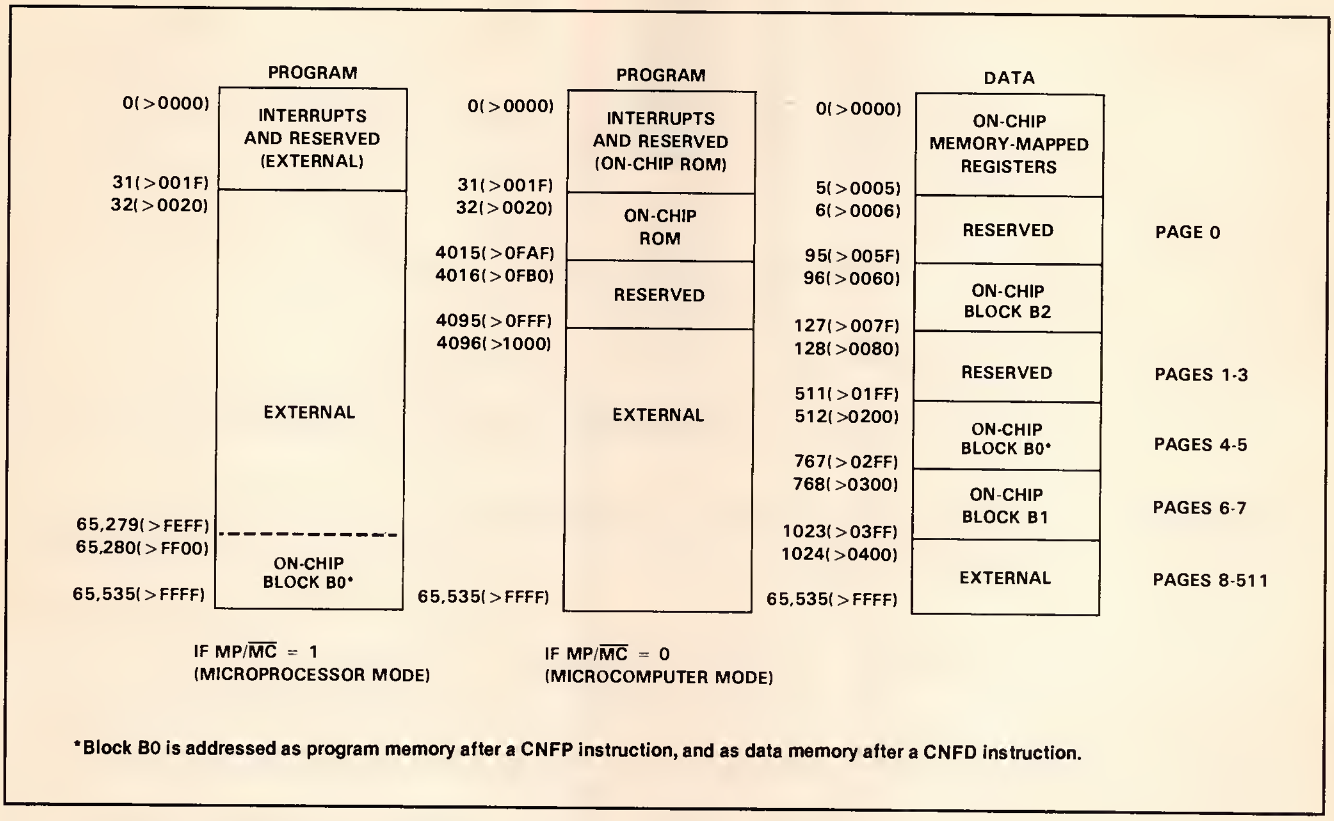
Two large, on-chip data RAM blocks (a total of 544 words), one of which is configurable either as program or data memory, are provided. An off-chip, 64K-word, directly addressable data memory address space is included to facilitate implementations of DSP algorithms with large data memory requirements. Four-K words of on-chip program ROM and 64K words of off-chip program address space are available. Large programs can execute at full speed from this memory space. Programs can also be



LEGEND:

ACCH	- Accumuletor high	IFR	-	Interrupt fleg register	PC		Progrem counter
ACCL	= Accumuletor low	IMR	-	Interrupt mesk register	PFC		Prefetch counter
ALU	= Arithmetic logic unit	IR	-	Instruction register	RPTC	-	Repeet instruction counter
ARAU	= Auxiliary register arithmetic unit	MCS	-	Microcall stack	GREG		Globel memory allocation register
ARB	= Auxiliery register pointer buffer	QIR		Queue instruction register	RSR		Serial port receive shift register
ARP	= Auxiliary register pointer	PR	-	Product register	XSR		Seriel port transmit shift register
DP	= Deta memory pege pointer	PRD	-	Period register for timer	AR0-AR7		Auxiliary registers
DRR	= Seriel port dete recaive register	TIM	-	Timer	ST0,ST1		Stetus registers
DXR	- Seriel port date trensmit register	TR	-	Temporery register			

Figure 1. TMS320C25 block diagram.



*Block B0 is addressed as program memory after a CNFP instruction, and as data memory after a CNFD instruction.

Figure 2. TMS320C25 memory maps.

downloaded from slow external memory to on-chip RAM for full-speed operation.

The TMS320C25 also incorporates a hardware timer and a block data transfer capability.

The diagram of the TMS320C25 in Figure 1 shows the principal blocks and data paths within the processor. It also shows all of the TMS320C25's interface pins.

The TMS320C25's architecture is built around the program and data buses. The program bus carries the instruction code and immediate operands from program memory. The data bus interconnects elements such as the central arithmetic logic unit (CALU) and the auxiliary register file to the data RAM. Together, the program and data buses can carry data from on-chip data RAM and internal or external program memory to the multiplier in a single cycle for multiply/accumulate operations.

A high degree of parallelism exists in the device—for example, while data are being operated on by the CALU, arithmetic operations can be implemented in the auxiliary register arithmetic unit (ARAU). Such parallelism results in a powerful set of arithmetic, logical, and bit-manipulation operations that can be performed in a single machine cycle.

Memory allocation. As mentioned above, the TMS320C25 provides 4K 16-bit words of on-chip program ROM and 544 16-bit words of on-chip data RAM. The RAM is divided into three blocks, B0, B1, and B2. Of the 544 words, 256 words (block B0) are configurable as either data memory or program memory; 288 words (blocks B1 and B2) are always data memory. A data memory size of 544 words allows the TMS320C25 to handle a data array of 512 words but still leaves 32 locations for intermediate storage.

The TMS320C25 maintains separate address spaces for program memory, data memory, and I/O. In addition to blocks B0, B1, and B2, the on-chip data memory map (see Figure 2) includes memory-mapped registers. Six peripheral registers, the serial-port registers (DRR and DXR), timer register (TIM), period register (PRD), interrupt mask register (IMR), and global memory allocation register (GREG), have been mapped into the data memory space so they can be easily modified.

The TMS320C25 has a register file containing eight auxiliary registers that can be used for indirect addressing of data memory or for temporary storage. These registers, AR0-AR7, can be either directly addressed by an instruction or indirectly addressed by a three-bit auxiliary register pointer (ARP). The auxiliary registers and the ARP can be loaded either from data memory or by an immediate operand defined in the instruction. The contents of the registers can also be stored in data memory.

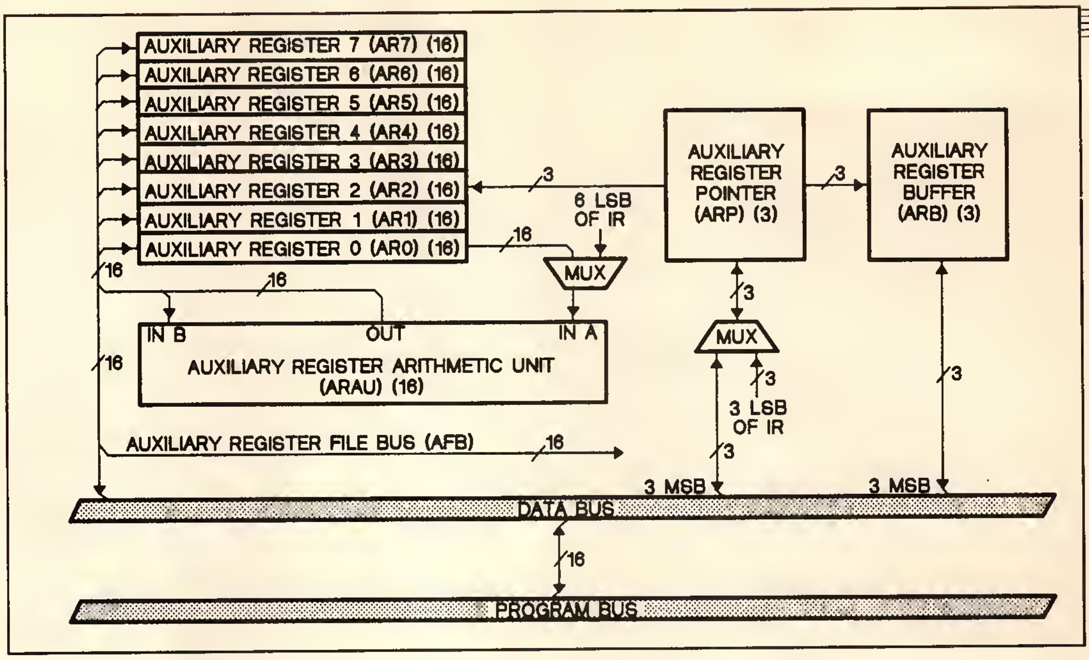

AUXILIARY REGISTER 7 (AR7) (16)
AUXILIARY REGISTER 6 (AR6) (16)
AUXILIARY REGISTER 5 (AR5) (16)
AUXILIARY REGISTER 4 (AR4) (16)
AUXILIARY REGISTER 3 (AR3) (16)
AUXILIARY REGISTER 2 (AR2) (16)
AUXILIARY REGISTER 1 (AR1) (16)
AUXILIARY REGISTER 0 (AR0) (16)
3
6 LSB OF IR
16
MUX
AUXILIARY REGISTER POINTER (ARP) (3)
3
AUXILIARY REGISTER BUFFER (ARB) (3)
16
16
IN B
OUT
IN A
3
MUX
16
AUXILIARY REGISTER ARITHMETIC UNIT (ARAU) (16)
3
3 LSB OF IR
3
AUXILIARY REGISTER FILE BUS (AFB)
16
3
3 MSB
3 MSB
DATA BUS
16
PROGRAM BUS

Figure 3. Auxiliary register file.

The auxiliary register file is connected to the auxiliary register arithmetic unit as shown in Figure 3. The ARAU can autoindex the current auxiliary register while the data memory location is being addressed. The current auxiliary register can also be indexed either by +1/−1 or by the contents of AR0. As a result, the accessing of tables of information does not require the CALU for address manipulation, thereby freeing it for other operations.

Although the ARAU was designed to support address manipulation in parallel with other operations, it can also serve as an additional general-purpose arithmetic unit since the auxiliary register file can communicate directly with data memory. The ARAU implements 16-bit unsigned arithmetic, whereas the CALU implements 32-bit two's-complement arithmetic. The ARAU also provides branches dependent on the comparison of AR0 to the auxiliary register pointed to by the ARP.

Central arithmetic logic unit. The CALU contains a 16-bit scaling shifter, a 16 × 16-bit parallel multiplier, a 32-bit ALU, and a 32-bit accumulator. The scaling shifter has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. This shifter produces a left shift of 0 to 16 bits on the input data, as programmed in the instruction. The least significant bits of the output are filled with zeroes, and the most significant bits are either filled with zeroes or sign-extended, depending upon the state of the sign-extension mode bit of status register ST1. Additional shifters at the outputs of both the accumulator and the multiplier are suitable for numerical scaling, bit extraction, extended-precision arithmetic, and overflow prevention. Due to the pipelining in the TMS320C25, shifting is accomplished as part of an instruction and thus does not require additional cycles for execution.

The 32-bit ALU and accumulator perform a wide range of arithmetic and logical instructions. An overflow saturation mode permits the accumulator to be loaded with the most positive or negative number (the choice depending on the direction of overflow), and it allows an overflow flag to be set whenever an overflow occurs. One of the two inputs to the ALU is always provided from the accumulator, and the other may be transferred from the product register (PR) of the multiplier or from the scaling shifter loaded from data memory.

The implementation of a typical ALU instruction requires these steps:

- data are fetched from the Rcarrn the data bus;
- data are passed through the scaling shifter and through the ALU, where the arithmetic is performed; and
- the result is moved into the accumulator.

The 32-bit accumulator is split into two 16-bit segments for storage in data memory: ACCH (accumulator high) and ACCL (accumulator low). Shifters at the output of the accumulator provide a shift of 0 to 7 places to the left. This shift is performed while the data are being transferred to the data bus for storage. The contents of the accumulator remain unchanged. The accumulator also has an in-place one-bit shift to the left or right (SFL or SFR instruction) and a rotate through carry (ROL or ROR instruction) for shifting its contents.

A carry bit is provided to the accumulator, allowing more efficient extended-precision computation. ADDC (add with carry) and SUBB (subtract with borrow) are two instructions using the carry bit. Branch instructions that use the carry bit are also provided.

Hardware multiplier. The TMS320C25 uses a 16 × 16-bit hardware multiplier that can compute a 32-bit product during every machine cycle. Two registers are associated with the multiplier: a 16-bit temporary register (TR) that holds one of the operands for the multiplier, and a 32-bit product register (PR) that holds the product.

The output of the product register can be left-shifted one or four bits. This is useful for implementing fractional arithmetic or justifying fractional products. The output of the PR can also be right-shifted six bits to enable the execution of up to 128 consecutive multiply/accumulates without overflow.

The multiplier performs both signed and unsigned operations. Two signed instructions, MAC (multiply/accumulate) and MACD (multiply/accumulate and data move), can process both operands simultaneously, thereby fully utilizing the computational bandwidth of the multiplier. For MAC and MACD, the two operands are transferred to the multiplier at each cycle via the program and data buses. This enables MAC and MACD to be performed in a single cycle when they are used with repeat (RPT or RPTK) instructions. The program bus can supply data from internal or external memory (RAM or ROM) and still maintain single-cycle operation. An unsigned multiply (MPYU) instruction facilitates extended-precision multiplication. It multiplies the unsigned contents of the TR by the unsigned contents of the addressed data memory location, and places the result in the PR.

Control operations. Control operations are provided on the TMS320C25 by an on-chip timer, a repeat counter, three external maskable user interrupts, and internal interrupts generated by serial-port operations or by the timer.

A memory-mapped 16-bit timer (TIM) register (a down counter) is continuously clocked by CLKOUT1. A timer interrupt (TINT) is generated whenever the timer decrements to zero. The timer is reloaded with the value contained in the period (PRD) register within the first cycle after it reaches zero so that interrupts may be programmed to occur at regular intervals of (PRD + 1) * CLKOUT1 cycles. This feature is useful for control operations and for synchronous sampling of or writing to peripherals.

The repeat counter (RPTC) is loaded with either a data memory value (in the case of the RPT instruction) or an immediate value (in the case of the RPTK instruction). The repeat feature enables a single instruction to be executed up to 256 times. It can be used with instructions such as multiply/accumulates, block moves, I/O transfers, and table read/writes. Those instructions that are normally multicycle are pipelined when the repeat feature is used and effectively become single-cycle instructions. For example, the table read (TBLR) instruction ordinarily takes three or more cycles, but when it is repeated, it becomes a single-cycle instruction.

The three external maskable user interrupts, $\overline{\mathrm{INT}}$2 to $\overline{\mathrm{INT}}$0, enable external devices to interrupt the processor. Internal interrupts are generated by either the serial port, the timer, or the software interrupt instruction. Interrupts are prioritized, with reset having the highest priority and the serial-port transmit interrupt the lowest.

Serial port. An on-chip serial port provides direct communication with serial devices such as codecs and serial A/D and D/A converters. The serial port's interface requires a minimum of external hardware. The port has two memory-mapped registers—a data transmit register and a data receive register—which can be operated in either an eight-bit byte mode or a 16-bit word mode. The transmit framing sync pulse can be generated internally or externally. The serial port's maximum speed is 5 MHz.

The primary enhancements of the TMS320C25's serial port are

- double buffering for both receive and transmit operations,
- the elimination of a minimum CLKR/CLKX frequency (fmin = 0 Hz), and
- the provision of a frame sync mode (FSM) bit, which allows continuous operation with no frame sync pulses.

The FSM is useful for communicating on pulse-code-modulated telephone system highways. As a result the TMS-320C25 can communicate directly on PCM highways such as AT&T T-1 and CCITT G.711/712 by counting the transmitted and received bytes in software and performing the instructions needed to set (SFSM) and reset (RFSM) the FSM bit.

I/O interface. The TMS320C25's I/O space consists of 16 input and 16 output ports. These ports provide a full 16-bit parallel I/O interface via the processor's data bus. A single input (IN) or output (OUT) operation typically takes two cycles; however, when executed in the repeat mode, such an operation becomes single-cycle. The TMS320C25 supports a range of system interfacing requirements. As previously mentioned, three separate address spaces—program, data, and I/O—provide interfacing to memory and I/O, thereby maximizing system throughput. The TMS320C25 simplifies I/O design by treating I/O the same way it treats memory. It maps I/O devices into the I/O address space using its external address and data buses in the same way as it uses them for mapping memory devices into memory address space.

The local memory interface consists of a 16-bit parallel data bus (D15-D0), a 16-bit address bus (A15-A0), three pins for data memory, program memory, and I/O space select ($\overline{\mathrm{DS}}$, $\overline{\mathrm{PS}}$, and $\overline{\mathrm{IS}}$, respectively), and various system control signals. The R/$\overline{\mathrm{W}}$ signal controls the direction of a data transfer, and $\overline{\mathrm{STRB}}$ provides a timing signal to control the transfer. When using on-chip program RAM, ROM, or high-speed external program memory, the TMS320C25 runs at full speed without wait states. By using the READY signal, it can generate wait states so it can communicate with slower off-chip memories.

The TMS320C25 supports direct memory access to external program and data memory. Another processor can take complete control of the TMS320C25's external memory by asserting $\overline{\mathrm{HOLD}}$ low, causing the TMS320C25 to place its address, data, and control lines in the high-impedance state. Two modes are available on the device. In the first mode, execution is suspended during assertion of $\overline{\mathrm{HOLD}}$. In the second mode—the "concurrent DMA mode"—the TMS-320C25 continues to execute its program while operating from internal RAM or ROM, thereby greatly increasing throughput in data-intensive applications. Signaling between the external processor and the TMS320C25 can be performed through interrupts.

Table 1.
TMS320C25 instructions.

ACCUMULATOR MEMORY REFERENCE INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	OPERATION
ABS	Absolute value of accumulator	1	$\lvert(ACC)\rvert$ → ACC
ADD	Add to accumulator with shift	1	(ACC) + [(dma) × 2^{shift}] → ACC
ADDC‡	Add to accumulator with carry	1	(ACC) + (dma) + (C) → ACC
ADDH	Add to high accumulator	1	(ACC) + [(dma) × 2^{16}] → ACC
ADDK‡	Add to accumulator short immediate	1	(ACC) + B-bit constant → ACC
ADDS	Add to low accumulator with sign extension suppressed	1	(ACC) + (dma) → ACC
ADDT†	Add to accumulator with shift specified by T register	1	(ACC) + [(dma) × $2^{(Treg)}$] → ACC
ADLK†	Add to accumulator long immediate with shift	2	(ACC) + [16-bit constant × 2^{shift}] → ACC
AND	AND with accumulator	1	(ACC(15-0)).AND.(dma) → ACC(15-0), 0 → ACC(31-16)
ANDK†	AND immediate with accumulator with shift	2	(ACC)30-0)).AND.[16-bit constant × 2^{shift}] → ACC(30-0), 0 → ACC(30-0)
CMPL†	Complement accumulator	1	($\overline{ACC}$) → ACC
LAC	Load accumulator with shift	1	(dma) × 2^{shift} → ACC
LACK	Load accumulator immediate short	1	B-bit constant → ACC
LACT†	Load accumulator with shift specified by T register	1	(dma) × $2^{(Treg)}$ → ACC
LALK†	Load accumulator long immediate with shift	2	(16-bit constant) × 2^{16} → ACC
NEG†	Negate accumulator	1	− (ACC) → ACC
NORM†	Normalize contents of accumulator	1	
OR	OR with accumulator	1	(ACC(15-0)).OR. (dma) → ACC(15-0)
ORK†	OR immediate with accumulator with shift	2	(ACC(30-0)).OR.[16-bit constant × 2^{shift}] → ACC(30-0)
ROL‡	Rotate accumulator left	1	(ACC(30-0)) → ACC(31-1), (C) → ACC(0), (ACC(31)) → C
ROR‡	Rotate accumulator right	1	(ACC(31-1)) → ACC(30-0), (C) → ACC(31), (ACC(0)) → C
SACH	Store high accumulator with shift	1	[(ACC) × 2^{shift}] → dma
SACL	Store low accumulator with shift	1	[(ACCL) × 2^{shift}] → dma
SBLK†	Subtract from accumulator long immediate with shift	2	(ACC) − [16-bit constant × 2^{shift}] → ACC
SFL†	Shift accumulator left	1	(ACC(30-0)) → ACC(31-1), 0 → ACC(0)
SFR†	Shift accumulator right	1	(ACC(31-1)) → ACC(30-0), (ACC(31)) → ACC(31)
SUB	Subtract from accumulator with shift	1	(ACC) [(dma) × 2^{shift}] → ACC
SUBB‡	Subtract from accumulator with borrow	1	(ACC) − (dma) − ($\overline{C}$) → ACC
SUBC	Conditional subtract	1	
SUBH	Subtract from high accumulator	1	(ACC) [(dma) × 2^{16}] → ACC
SUBK‡	Subtract from accumulator short immediate	1	(ACC) − 8-bit constant → ACC
SUBS	Subtract from low accumulator with sign extension suppressed	1	(ACC) − (dma) → ACC
SUBT†	Subtract from accumulator with shift specified by T register	1	(ACC) − [(dma) × $2^{(Treg)}$] → ACC
XOR	Exclusive-OR with accumulator	1	(ACC(15-0)).XOR.(dma) → ACC(15-0)
XORK†	Exclusive-OR immediate with accumulator with shift	2	(ACC(30-0)).XOR.[16-bit constant × 2^{shift}] → ACC(30-0)
ZAC	Zero accumulator	1	0 → ACC
ZALH	Zero low accumulator and load high accumulator	1	(dma) × 2^{16} → ACC
ZALR‡	Zero low accumulator and load high accumulator with rounding	1	(dma) × 2^{16} + >B000 → ACC
ZALS	Zero accumulator and load low accumulator with sign extension suppressed	1	(dma) → ACCL, 0 → ACCH

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

AUXILIARY REGISTERS AND DATA PAGE POINTER INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	OPERATION
ADRK‡	Add to auxiliary register short immediate	1	(ARn) + 8-bit constant → ARn
CMPR†	Compare auxiliary register with auxiliary register AR0	1	If ARn | CM | AR0, then 1 → TC; else 0 → TC
LAR	Load auxiliary register	1	(dma) → (ARn)
LARK	Load auxiliary register short immediate	1	8-bit constant → ARn
LARP	Load auxiliary register pointer	1	3-bit constant → ARP, (ARP) → ARB
LDP	Load data memory page pointer	1	(dma) → DP
LDPK	Load data memory page pointer immediate	1	9-bit constant → DP
LRLK†	Load auxiliary register long immediate	2	16-bit constant → ARn
MAR	Modify auxiliary register	1	
SAR	Store auxiliary register	1	(ARn) → dma
SBRK‡	Subtract from auxiliary register short immediate	1	(ARn) – 8-bit constant → ARn

T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	OPERATION
APAC	Add P register to accumulator	1	(ACC) + (shift Preg) → ACC
LPH†	Load high P register	1	(dma) → Preg (31-16)
LT	Load T register	1	(dma) → Treg
LTA	Load T register and accumulate previous product	1	(dma) → Treg, (ACC) + (shifted Preg) → ACC
LTD	Load T register, accumulate previous product, and move data	1	(dma) → Treg, (dma) → dma + 1, (ACC) + (shifted Preg) → ACC
LTP†	Load T register and store P register in accumulator	1	(dma) → Treg, (shifted Preg) → ACC
LTS†	Load T register and subtract previous product	1	(dma) → Treg, (ACC) – (shifted Preg) → ACC
MAC†	Multiply and accumulate	2	(ACC) + (shifted Preg) → ACC, (pma) × (dma) → Preg
MACD†	Multiply and accumulate with data move	2	(ACC) + (shifted Preg) → ACC, (pma) × (dma) → Preg, (dma) → dma + 1
MPY	Multiply (with T register, store product in P register)	1	(Treg) × (dma) → Preg
MPYA‡	Multiply and accumulate previous product	1	(ACC) + (shifted Preg) → ACC, (Treg) × (dma) → Preg
MPYK	Multiply immediate	1	(Treg) × 13-bit constant → Preg
MPYS‡	Multiply and subtract previous product	1	(ACC) – (shifted Preg) → ACC, (Treg) × (dma) → Preg
MPYU‡	Multiply unsigned	1	Usgn (Treg) × Usgn (dma) → Preg
PAC	Load accumulator with P register	1	(shifted Preg) → ACC
SPAC	Subtract P register from accumulator	1	(ACC) – (shifted Preg) → ACC
SPH‡	Store high P register	1	(shifted Preg (31-16)) → dma
SPL‡	Store low P register	1	(shifted Preg (15-0)) → dma
SPM†	Set P register output shift mode	1	2-bit constant → PM
SQRA†	Square and accumulate	1	(ACC) + (shifted Preg) → ACC, (dma) × (dma) → Preg
SQRS†	Square and subtract previous product	1	(ACC) – (shifted Preg) → ACC, (dma) × (dma) → Preg

SYMBOL	MEANING
ACC	Accumulator
ARB	Auxiliary register pointer buffer
ARn	Auxiliary Register n (AR0 through AR7 are predefined assembler symbols equal to 0 through 7 respectively)
ARP	Auxiliary register pointer
BIO	Branch control input
C	Carry bit
CM	2 bit field specifying compare mode
CNF	On-chip RAM configuration control bit
dma	Data memory address
DP	Data page pointer
FO	Format status bit
FSM	Frame synchronization mode bit
HM	Hold mode bit
INTM	Interrupt mode flag bit
>nn	Indicates nn is a hexadecimal number (All others are assumed to be decimal values)
OV	Overflow flag bit
OVM	Overflow mode bit
P	Product register

SYMBOL	MEANING
PA	Port address (PA0 through PA15 are predefined assembler symbols equal to 0 through 15 respectively)
PC	Program counter
PM	2 bit field specifying P register output shift code
pma	Program memory address
Preg	Product register
RPTC	Repeat counter
STn	Status Register n (ST0 or ST1)
SXM	Sign extension mode bit
T	Temporary register
TC	Test control bit
TOS	Top of stack
Treg	Temporary register
TXM	Transmit mode bit
Usgn	Unsigned value
XF	XF pin status bit
→	Is assigned to
	An absolute value
[ ]	Optional items
( )	Contents of

Table 1 cont'd

BRANCH/CALL INSTRUCTIONS

MNEMONIC	DESCRIPTION	NO. WORDS	OPERATION
B	Branch unconditionally	2	pma → PC
BACC†	Branch to address specified by accumulator	1	(ACC(15-0)) → PC
BANZ	Branch on auxiliary register not zero	2	If (AR(ARP)) ≠ 0, then pma → PC; else (PC) + 2 → PC
BBNZ†	Branch if TC bit ≠ 0	2	If (TC) = 1, then pma → PC; else (PC) + 2 → PC
BBZ†	Branch if TC bit = 0	2	If (TC) = 0, then pma → PC; else (PC) + 2 → PC
BC‡	Branch on carry	2	If (C) = 1, then pma → PC; else (PC) + 2 → PC
BGEZ	Branch if accumulator ≥ 0	2	If (ACC) ≥ 0, then pma → PC; else (PC) + 2 → PC
BGZ	Branch if accumulator > 0	2	If (ACC) > 0, then pma → PC; else (PC) + 2 → PC
BIOZ	Branch on I/O status = 0	2	If ($\overline{\text{BIO}}$) = 0, then pma → PC; else (PC) + 2 → PC
BLEZ	Branch if accumulator ≤ 0	2	If (ACC) ≤ 0, then pma → PC; else (PC) + 2 → PC
BLZ	Branch if accumulator < 0	2	If (ACC) < 0, then pma → PC; else (PC) + 2 → PC
BNC‡	Branch on no carry	2	If (C) = 0, then pma → PC; else (PC) + 2 → PC
BNV†	Branch if no overflow	2	If (OV) ≠ 0, then pma → PC; else (PC) + 2 → PC
BNZ	Branch if accumulator ≠ 0	2	If (ACC) ≠ 0, then pma → PC; else (PC) + 2 → PC
BV	Branch on overflow	2	If (OV) = 0, then pma → PC; else (PC) + 2 → PC
BZ	Branch if accumulator = 0	2	If (ACC) = 0, then pma → PC; else (PC) + 2 → PC
CALA	Call subroutine indirect	1	(ACC(15-0)) → PC, (PC) + 1 → TOS
CALL	Call subroutine	2	(PC) + 2 → TOS, pma → PC
RET	Return from subroutine	1	(TOS) → PC

I/O AND DATA MEMORY OPERATIONS

MNEMONIC	DESCRIPTION	NO. WORDS	OPERATION
BLKD†	Block move from data memory to data memory	2	(dma1, addressed by PC) → dma2
BLKP†	Block move from program memory to data memory	2	(pma, addressed by PC) → dma
DMOV	Data move in data memory	1	(dma) → dma + 1
FORT†	Format serial port registers	1	1-bit constant → FO
IN	Input data from port	1	(data bus, addressed by PA) → dma
OUT	Output data to port	1	(dma) → data bus, addressed by PA
RFSM‡	Reset serial port frame synchronization mode	1	0 → FSM
RTXM†	Reset serial port transmit mode	1	0 → TXM
RXF†	Reset external flag	1	0 → XF
SFSM‡	Set serial port frame synchronization mode	1	1 → FSM
STXM†	Set serial port transmit mode	1	1 → TXM
SXF†	Set external flag	1	1 → XF
TBLR	Table read	1	(pma, addressed by ACC (15-0)) → dma
TBLW	Table write	1	(dma) → pma, addressed by ACC (15-0)

CONTROL INSTRUCTIONS

MNEMONIC	DESCRIPTION	NO. WORDS	OPERATIONS
BIT†	Test bit	1	(dma bit at (15-bit code)) → TC
BITT†	Test bit specified by T register	1	(dma bit at (15-Treg)) → TC
CNFD†	Configure block as data memory	1	0 → CNF
CNFP†	Configure block as program memory	1	1 → CNF
DINT	Disable interrupt	1	1 → INTM
EINT	Enable interrupt	1	0 → INTM
IDLE†	Idle until interrupt	1	(PC) + 1 → PC, powerdown
LST	Load status register ST0	1	(dma) → ST0
LST1†	Load status register ST1	1	(dma) → ST1

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

CONTROL INSTRUCTIONS cont'd			
NOP	No operation	1	(PC) + 1 → PC
POP	Pop top of stack to low accumulator	1	(TOS) → ACC
POPD†	Pop top of stack to data memory	1	(TOS) → dma
PSHD†	Push data memory value onto stack	1	(dma) → TOS
PUSH	Push low accumulator onto stack	1	(ACCL) → TOS
RC‡	Reset carry bit	1	0 → C
RHM‡	Reset hold mode	1	0 → HM
ROVM	Reset overflow mode	1	0 → OVM
RPT†	Repeat instruction as specified by data memory value	1	(dma) → RPTC
RPTK†	Repeat instruction as specified by immediate value	1	8-bit constant → RPTC
RSXM†	Reset sign-extension mode	1	0 → SXM
RTC‡	Reset test/control flag	1	0 → TC
SC‡	Set carry bit	1	1 → C
SHM‡	Set hold mode	1	1 → HM
SOVM	Set overflow mode	1	1 → OVM
SST	Store status register ST0	1	ST0 → dma
SST1†	Store status register ST1	1	ST1 → dma
SSXM†	Set sign-extension mode	1	1 → SXM
STC‡	Set test/control flag	1	1 → TC
TRAP†	Software interrupt	1	(PC) + 1 → TOS, 30 → PC

The TMS320C25's conditions and modes are stored in two status registers, ST0 and ST1. Instructions are provided to allow these registers to be stored in or loaded from data memory. This capability allows the current status of the device to be saved during interrupts and subroutine calls.

TMS320C25 software

Earlier, we characterized digital signal processing as the real-time processing of mathematically intensive algorithms. This characterization equates to a requirement for high-speed, multiply/accumulate capability in a processor. The performance of a signal processor is therefore measured in terms appropriate to this requirement—that is, it is measured in terms of the speed of execution of individual instructions, the power of the instruction set, and the I/O capabilities. The speed is given as the basic instruction cycle time and the number of cycles required to complete any instruction.

As we noted earlier, pipelining of instruction fetching, decoding, and execution provides an instruction cycle time of only 100 ns. The overwhelming majority of the TMS320C25's instructions (97 out of 133) are executed in a single instruction cycle. Of the 36 instructions requiring additional cycles for execution, 21 involve branches, calls, and returns that result in a reload of the program counter and a break in the execution pipeline. Another seven of the instructions are two-word, long immediate instructions. The remaining eight—IN, OUT, BLKD, BLKP, TBLR, TBLW, MAC, and MACD—support I/O and transfers of data between memory spaces, or provide for additional parallel operation in the processor. Furthermore, these eight instructions become single-cycle when used in conjunction with the repeat counter. The instruction set of the TMS320C25 exploits the parallelism of the processor, allowing complex or numerically intensive computations to be implemented in relatively few instructions. Table 1 lists the TMS320C25's instructions.

Addressing modes. Most TMS320C25 instructions are coded in a single 16-bit word—the reason most can be executed in a single cycle. The 16-bit word comprises an eight-bit opcode and an eight-bit address. Three memory addressing modes are available: direct, indirect, and immediate (Table 2). Both direct and indirect addressing are used to access data memory. Immediate addressing uses the contents of the memory addressed by the program counter. Figure 4 illustrates operand addressing in the direct, indirect, and immediate modes.

In direct addressing, seven bits of the instruction word are concatenated with the nine-bit data memory page pointer (DP) to form the 16-bit data memory address. The DP register points to one of 512 possible data memory pages, each 128 word in length, to obtain a 64K total data memory space. The seven-bit address in the instruction

Table 2.
Addressing modes.

ADDRESSING MODE	OPERATION
OP A	Direct addressing
OP *(,NARP)	Indirect; no change to AR.
OP *+(,NARP)	Indirect; current AR is incremented.
OP *-(,NARP)	Indirect; current AR is decremented.
OP *0+(,NARP)	Indirect; AR0 is added to current AR.
OP *0-(,NARP)	Indirect; AR0 is subtracted from current AR.
OP *BR0+(,NARP)	Indirect; AR0 is added to current AR (with reverse carry propagation).
OP *BR0-(,NARP)	Indirect; AR0 is subtracted from current AR (with reverse carry propagation).

NOTE: The optional NARP field specifies a new value of the ARP.

points to the specific location within the data memory page.

Indirect addressing is provided by the eight auxiliary registers AR0-AR7. These registers can be used to indirectly address data memory, as loop counters, or for temporary data storage. Indirect auxiliary register addressing (Figure 5) allows placement of the data memory address of an instruction operand into one of the eight auxiliary registers. These registers are pointed to by a three-bit auxiliary register pointer (ARP) that is loaded with a value from 0 through 7 designating AR0 through AR7, respectively. The auxiliary registers and the ARP may be loaded either from data memory or by an immediate operand defined in the instruction. Furthermore, the contents of the auxiliary registers may be stored in data memory.

There are seven types of indirect addressing (see Table 2 again):

- indexing with increment,
- indexing with decrement,
- indexing by adding the contents of AR0,
- indexing by subtracting the contents of AR0,
- indexing by adding the contents of AR0 with the carry propagation reversed (for bit-reversing an FFT),
- indexing by subtracting the contents of AR0 with the carry propagation reversed (also for bit-reversing an FFT), and
- no indexing.

All indexing operations are performed on the current auxiliary register in the same cycle as the original instruction, with loading of a new ARP value available as an option. The operations performed in the ARAU can even be performed during branch instruction execution, allowing efficient control with conditional looping.

Bit-reversed indexed addressing modes allow efficient I/O to be performed for the resequencing of data points in a radix-2 FFT program. The direction of carry propagation in the ARAU is reversed when this mode is selected, and AR0 is added to or subtracted from the current auxiliary register.

In immediate addressing, the instruction word contains the value of the immediate operand. Both single-word (8-bit and 13-bit constant) short immediate instructions and two-word (16-bit constant) long immediate instructions are included in the instruction set. In the case of long immediate instructions, the word following the instruction opcode is used as the immediate operand. MPYK is an example of an immediate instruction; it multiplies the contents of the T register by a signed 13-bit constant. Seventeen immediate operand instructions are included in the instruction set (see Table 1 again).

Instruction set parallelism—an example. The MACD (multiply/accumulate and data move) instruction serves as an informative example of the parallelism designed into the TMS320C25 instruction set as well as into the TMS320C25 architecture. As shown in Equation 1, the requirement for parallelism exists in common DSP operations such as convolution and filtering.[6,7]

Parallelism in the execution of instructions enables a complete multiply/accumulate/data move operation to be completed in a single 100-ns instruction cycle. The execution of the MACD involves the following steps:

1) The contents of the 32-bit P register are shifted (scaled) by an output shifter.

2) The 32-bit ALU accumulates the shifted result of the 32-bit P register with the current contents of the 32-bit accumulator.

3) The 16-bit contents of a data memory location (usually addressed indirectly via one of the auxiliary registers) are loaded into the T register.

4) The 16-bit contents of a program memory location (addressed via the prefetch counter PFC) are introduced to the multiplier and a 16 × 16-bit multiply is executed, resulting in a new 32-bit product. The product is placed in the P register to be accumulated during the next cycle.

5) The 16-bit contents of the data memory location are copied to the next higher data memory address.

6) The carry and overflow status bits are set, as appropriate, in the status registers.

7) The 16-bit contents of the auxiliary register pointed to by the ARP are modified (typically decremented) in preparation for the use of the data memory address on the next cycle.

8) The 16-bit contents of the PFC are incremented in preparation for the use of the program memory address on the next cycle.

9) The repeat counter is decremented.

As can be seen from the above, one of the data values is taken from data memory while the other is taken from program memory. A single-cycle execution and data move is accomplished when the data memory being addressed is the on-chip data memory. The program memory location can be either on or off chip and, if on chip, can come from either ROM or the reconfigurable memory block B0.

Parallel operation of certain subsets of TMS320C25 functions is also available. These subsets include loading the T register in combination with addition (LTA), subtraction (LTS), or a move of the P register's contents to the accumulator (LTP). The accumulation can be supplemented by the data move function (LTD). Another combination (MPYA/MPYS) provides the accumulation of the previous

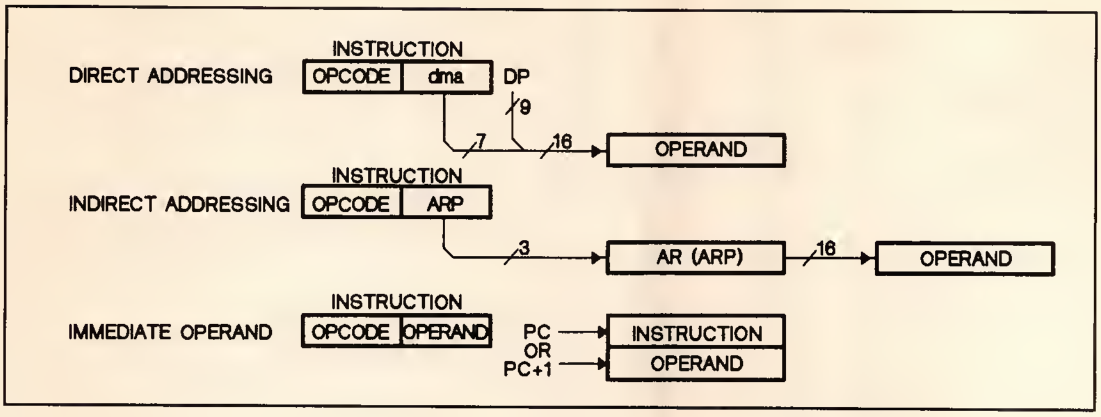


Figure 4. Methods of addressing the instruction operand.

product along with the execution of the multiplier to generate a new product. This combination is particularly useful in adaptive filtering techniques such as those embodied in the least-mean-square (LMS) algorithm.[4,15] The implementation of an adaptive filter by means of these instructions will be described in detail in the section on applications.

Block moves. The TMS320C25 provides six instructions for data and program block moves and transfers of data via the I/O ports. When these instructions are pipelined by means of the repeat instruction, significantly higher throughput is achieved—the pipelining results in a transfer rate of 160 million bits per second.

The BLKD instruction moves a block within data memory, and the BLKP instruction moves a block from program memory to data memory. Block transfers between program and data memory spaces can also be implemented with the TBLR and TBLW (table read and table write) instructions. The advantages of TBLR and TBLW are that they allow the source address as well as the destination address to be determined during programming and that they permit the data to be transferred from data memory to program memory. The IN and OUT instructions permit data to be transferred between the I/O and data memory spaces. While the source address is determined by the prefetch counter, which is incremented on every cycle, the destination address is determined by an auxiliary register whose contents can be modified in any of the previously specified ways. This permits sequential and contiguous data placement (*+, *−), sequential but noncontiguous data placement (*0+, *0−), or scrambled data placement (*BR0+, *BR0−). The value of these address modifications during block data transfers becomes particularly apparent in the use of indexing with reverse-carry propagation to set up the data block in an FFT. The result is not only a savings in execution time but a savings in program memory space as well.

Floating-point support. The TMS320C25 supports floating-point operations for applications requiring a large dynamic range. The NORM (normalization) instruction normalizes fixed-point numbers contained in the accumulator or by performing left shifts. The LACT (load accumulator with shift specified by the T register) instruction denormalizes a floating-point number by arithmetically left-shifting the mantissa through the input scaling shifter. The shift count, in this case, is the value of the exponent specified by the four low-order bits of the T register. ADDT and SUBT instructions (add to/subtract from accumulator with shift specified by the T register) have been provided to allow additional arithmetic operations.

TMS320C25 hardware

The most important task for a hardware designer is interfacing the DSP device to the rest of the system as inexpensively as possible. Here, we will discuss the TMS320C25's interfacing capabilities.

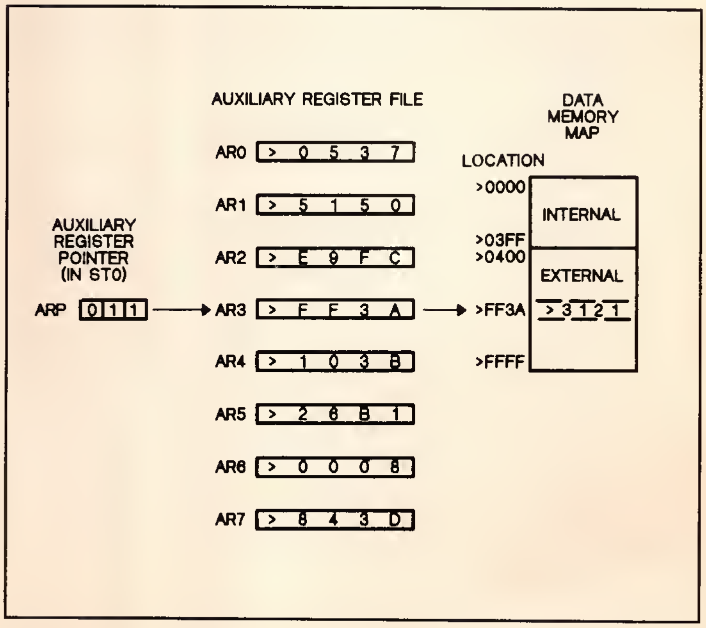


Figure 5. Example of indirect auxiliary register addressing.

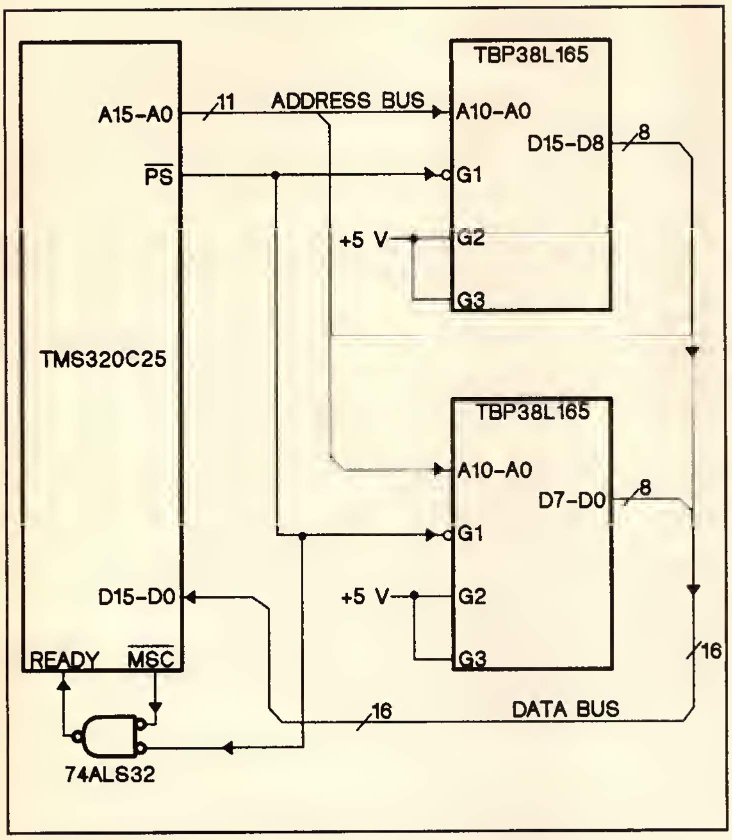


Figure 6. Minimal configuration for external program memory.

System configurations. The flexibility of the TMS320C25 allows systems configurations that satisfy a broad range of application requirements. The TMS320C25 can be configured as

- a stand-alone system (that is, as a single processor using 4K words of on-chip ROM and 544 words of on-chip RAM),
- part of a parallel multiprocessing system (two or more TMS320C25s) with shared global data memory, or
- a coprocessor for a host processor.

The stand-alone system interface consists of a 16-bit parallel data bus, a 16-bit address bus, three pins for memory space select, and various system control signals. In Figure 6, an external data RAM and a PROM/EPROM have been added to the basic stand-alone system. The READY signal is used for wait-state generation for communicating with slower off-chip memories. All the memories and I/O devices are directly controlled by the TMS320C25, thus minimizing external hardware requirements.

Parallel multiprocessing and host/coprocessor systems take advantage of the TMS320C25's direct memory access and global memory configuration capabilities.

Direct memory access. The TMS320C25 supports direct memory access to its external program/data memory and I/O space through its $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ signals. Direct memory access can be used for multiprocessing: Execution on one or more processors can be temporarily halted to allow another processor to read from or write to the halted processor's local off-chip memory. Here the multiprocessing is typically performed in a master/slave configuration. The master can initialize the slave by downloading a program into its program memory space or provide the slave with the data needed to complete a task.

In a direct memory access scheme, the master may be a general-purpose CPU, a TMS320C25, or perhaps even an A/D converter. A master TMS320C25 takes complete control of the slave's external memory by asserting $\overline{\text{HOLD}}$ low through its external flag (XF). This causes the slave to place its address, data, and control lines in a high-impedance state. By asserting $\overline{\text{RS}}$ in conjunction with $\overline{\text{HOLD}}$, the master processor can load the slave's local program memory with the necessary initialization code on reset or power-up. The two processors can be synchronized through use of the $\overline{\text{SYNC}}$ pin to make the transfer over the memory bus faster and more efficient.

After control of the slave's buses is given to the master processor, the slave alerts the master by asserting $\overline{\text{HOLDA}}$. This signal can be tied to the master's $\overline{\text{BIO}}$ pin. The slave's XF pin can be used to indicate to the master when the slave has finished performing its task and needs to be reprogrammed or given additional data to continue processing. In a multiple-slave configuration, the priority of each slave's task can be determined by tying the slave's XF signals to the appropriate $\overline{\text{INT}}$ pin on the master.

A PC environment provides an example of a direct memory access scheme in which the system bus is used for data transfer. In this configuration, either the master CPU or a disk controller may place data on the system bus for downloading into the local memory of the TMS320C25. Here the TMS320C25 acts like a peripheral processor with multifunction capability. In a speech application, for example, the master can load the TMS320C25's program memory with algorithms to perform tasks such as speech analysis, synthesis, or recognition, and its data memory with the required speech templates. In a graphics application, the TMS320C25 can serve as a dedicated graphics engine. programs can be stored in ROM or downloaded via the system bus into program RAM. Again, data can come from PC disk storage or be provided directly by the master CPU. In this configuration, decode and arbitration logic is used to control the direct memory access. When the address on the system bus resides in the local memory of the peripheral TMS320C25, this logic asserts the $\overline{\text{HOLD}}$ signal while sending the master a not-ready indication to allow wait states. After the TMS320C25 acknowledges the direct memory access by asserting $\overline{\text{HOLDA}}$, READY is asserted and the information is transferred.

Global memory. In some digital signal processing tasks, the algorithm being implemented can be divided into sections and a processor dedicated to each. In this case, the first and second processors can share global data memory, as can the second and third, the third and fourth, and so on. Arbitration logic may be required to determine which section of the algorithm will execute and which processor will have access to the global memory. The dedication of each processor to a distinct section of the algorithm makes pipelined execution—and thus higher throughput—possible.

External memory can be divided into global and local sections. Special registers and pins on the TMS320C25 allow multiple processors to share up to 32K words of global data memory. This facilitates efficient "shared data" multiprocessing, in which data are transferred between two or more processors. Unlike a direct memory access scheme, reading or writing global memory does not require one of the processors to be halted.

TMS320C25 development tools and support

A digital signal processor is essentially an application-specific microprocessor or microcomputer. Like any microprocessor, it needs good development tools and technical support—no matter how impressive its performance or how easy its interfacing to other devices, it cannot be easily designed into systems without such tools and support. In developing an application, a designer encounters problems and needs to ask questions. Often the tools and vendor support given him are the difference between the success and failure of his project.

The TMS320C25 is supported by many development tools.[16] These tools range from inexpensive modules for application evaluation and benchmarking to an assembler/linker and software simulator to a full-capability hardware emulator.

Software tools. An assembler/linker and software simulator that enable users to develop and debug TMS320 DSP algorithms are available for the TI PC, IBM PC, and VAX.

Assembler/linker. The macro assembler translates assembly language source code into executable object code. It allows the programmer to work with mnemonics rather than hexadecimal machine instructions and to reference memory locations with symbolic addresses. It supports macro calls and definitions along with conditional assembly. The linker permits a program to be designed and implemented in separate modules that are later linked to form the complete program. The linker resolves external definitions and references for relocatable code, creating an object file that can be executed by the simulator, emulator, or the TMS-320C25 processor. The macro assembler/linker is currently available for the VAX/VMS, TI PC/MS-DOS, and IBM PC/PC-DOS operating systems.

Simulator. The simulator is a software program that simulates TMS320 operations to allow program verification. Its debug mode enables the user to monitor the state of the simulated TMS320 while his program is executing. The simulator uses the object code produced by the macro assembler/linker. During program execution, the internal registers and memory of the simulated TMS320 are modified as each instruction is interpreted by the host computer. Once program execution is suspended, the internal registers and the program and data memories can be inspected and modified. The simulator is currently available for the VAX/VMS, TI PC/MS-DOS, and IBM PC/PC-DOS operating systems.

Hardware tools. Tools are provided for in-circuit emulation and hardware program debugging such as breakpointing and tracing so that DSP algorithms can be developed and tested in a real-product environment.

Evaluation module. The evaluation module, or EVM, is a stand-alone board that contains all the hardware tools needed to evaluate the TMS320C25 and that provides in-circuit emulation of it. The EVM's firmware package contains a debug monitor, an editor, an assembler, a reverse assembler, and software communication to two EIA ports. These ports allow the EVM to be connected to a terminal and to either a host computer or a line printer. The EVM accepts either source or object code downloaded from the host computer. Its resident assembler converts incoming source text into executable code in just one pass by automatically resolving labels after the first assembly pass is completed. When a session is finished, code is saved via the host computer interface.

Software development system. The SWDS is a plug-in card for the TI PC and IBM PC that provides the same functionality as the EVM.

Emulator. The XDS (Extended Development System) is an emulator providing full-speed in-circuit emulation with real-time hardware breakpointing and tracing and program execution capability from target memory. The XDS allows integration of hardware and software modules in the debug mode. By setting breakpoints based on internal conditions

No matter how impressive its performance or how easy its interfacing to other devices, a digital signal processor cannot be designed into systems without good development tools and vendor support.

or external events, the XDS user can suspend execution of the program and give control to the debug mode. In the debug mode, he can inspect and modify all registers and memory locations. Single-step execution is available. Full-trace capabilities at full speed and a reverse assembler that translates machine code back into assembly instructions also increase debugging productivity. The XDS system is designed to interface with either a terminal or a host computer. Object code generated by the assembler/linker can be downloaded to the XDS and then controlled through a terminal.

Analog interface board. The AIB is an analog-to-digital (A/D) and digital-to-analog (D/A) conversion board that can be used in conjunction with the EVM or XDS. It can also be used in an educational environment to help familiarize the user with real-world digital signal processing techniques. The AIB includes A/D and D/A converters with 12-bit resolution as well as antialiasing and smoothing filters that have a cut-off frequency programmable from 4.7 kHz to 20 kHz.

In addition to the above design tools, development support includes

- the Digital Filter Design Package, which runs on both TI and IBM PCs and which allows the user to design digital filters (low-pass, high-pass, band-pass, and band-stop types) using a menu-driven approach,
- TI Regional Technology Centers staffed with qualified engineers who provide technical support and design services,
- access to third parties with DSP expertise in various application areas,
- a series of DSP books covering DSP theory, algorithms, and applications and TMS320 implementations,[4,5,7]
- documentation such as user's guides,[10-12] data sheets, a development support reference guide,[16] and comprehensive application reports,[4] and
- a technical support hotline and a bulletin board service.

TMS320C25 applications

The TMS320C25 is designed for real-time DSP and other computation-intensive tasks in telecommunications, graphics, image processing, high-speed control, speech processing, instrumentation, and numeric processing. In these applications, the TMS320C25 provides an excellent means for executing signal processing algorithms such as fast Fourier transforms (FFTs), digital filters, frequency synthesizers, correlators, and convolution routines. It can also execute general-purpose functions since it includes bit-manipulation instructions, block data move capabilities, large program and data memory address spaces, and flexible memory mapping.

Since digital filters are used in so many DSP applications, let us examine them as a prelude to our discussion of TMS320C25 applications.

Digital filtering. Filters are often implemented in digital signal processing systems. Such filters fall into two categories: finite impulse response (FIR) filters and infinite impulse response (IIR) filters.[4,6] For both types of filter, the coefficients of the filter (weighting factors) may be fixed or adapted during the course of the signal processing. The TMS320C25 reduces the execution time of all filters by virtue of its 100-ns instruction cycle time and optimized instructions for filter operations.

As we stated earlier, the FIR filter is simply the sum of products in a sampled data system (see Equation 1 again). A simple implementation of the FIR filter uses the MACD instruction (multiply/accumulate and data move) for each filter tap and the RPT/RPTK instruction to repeat the MACD for each tap. Thus, a 256-tap FIR filter can be implemented as

```
RPTK  255
MACD  *-,COEFFP
```

Here, the coefficients can be stored anywhere in program memory (in the reconfigurable on-chip RAM, in the on-chip ROM, or in external memories). When the coefficients are stored in on-chip ROM or externally, the entire on-chip data RAM can be used to store the sample sequence. This allows filters of up to 512 taps to be implemented. Execution of the filter will be at full speed, or 100 ns per tap, as long as the memory (either on-chip RAM or high-speed external RAM) supports full-speed execution.

Up to this point, we have assumed that the filter coefficients are fixed from sample to sample. If the coefficients are adapted or updated with time, as they are in adaptive filters for echo cancellation,[4,15] the DSP algorithm requires a greater computational capacity from the processor. To adapt or update the coefficients, usually with each sample, the TMS320C25 uses three instructions—multiply and add/substract previous product to/from accumulator (MPYA/MPYS), zero-out low-order accumulator bits and load high-order accumulator bits with data (ZALR), and store high-order bits of accumulator to data memory (SACH). The method it uses to adapt the coefficients is the least-mean-square, or LMS, algorithm, which can be expressed as

$$b_k(i+1) = b_k(i) + 2\mathrm{B}\,[e(i) \cdot x(i-k)], \qquad (2)$$

where $b_k(i+1)$ is the weighting coefficient for the next sam ple period, $b_k(i)$ is the weighting coefficient for the present sample period, B is the gain factor or adaptation step size, $e(i)$ is the error function, and $x(i-k)$ is the input of the filter.

In an adaptive filter, the coefficients $b_k(i)$ must be updated to minimize the error function $e(i)$, which is the difference between the output of the filter and a reference signal. Quantization errors arising during coefficient updating can strongly affect the performance of the filter, but these errors can be minimized if the updated values are obtained by rounding rather than truncating. For each coefficient in the filter at a given point in time, the factor

$2*B*e(i)$ is a constant. This factor can be computed once and stored in the T register for each of the updates. This reduces the computational requirement to one multiply/accumulate plus rounding. Without the new instructions, the adaptation of each coefficient would take five instructions corresponding to five clock cycles, as the following instruction sequence shows:

```
LRLK   AR2,COEFFD ; LOAD ADDRESS OF COEFFICIENTS.
LRLK   AR3,LASTAP ; LOAD ADDRESS OF DATA SAMPLES.
LARP   AR2
LT     ERRF       ; errf = 2*B*e(i)
  .
  .
  .
ZALH   *,AR3      ; ACC = bk(i)*2**16
ADD    ONE,15     ; ACC = bk(i)*2**16 + 2**15
MPY    *-,AR2
APAC              ; ACC = bk(i)*2**16
                    + errf*x(i-k) + 2**15

SACH   *+         ; SAVE bk(i+1).
  .
  .
  .
```

When the MPYA and ZALR instructions are used, the adaptation reduces to three instructions corresponding to three clock cycles, as shown below:

```
LRLK   AR2,COEFFD ; LOAD ADDRESS OF COEFFICIENTS.
LRLK   AR3,LASTAP ; LOAD ADDRESS OF DATA SAMPLES.
LARP   AR2
LT     ERRF       ; errf = 2*B*e(i)
  .
  .
  .
ZALR   *,AR3      ; ACC = bk(i)*2**16 + 2**15
MPYA   *-,AR2     ; ACC = bk(i)*2**16
                    + errf*x(i-k) + 2**15
                  ; PREG = errf*x(i-k+1)
SACH   *+         ; SAVE bk(i+1).
  .
  .
  .
```

Note that the processing order has been slightly changed to incorporate the use of the MPYA instruction. This is due to the fact that the accumulation performed by the MPYA is the accumulation of the previous product.

We have now seen the basic code for a FIR filter tap and a coefficient update. Figure 7 shows a routine to filter a signal and update the coefficients for a 256-tap adaptive FIR filter. Note that for each tap one instruction cycle is needed to perform the FIR filter (i.e., to execute a MACD), three instruction cycles are needed to update the filter coefficients, and 33 instruction cycles are needed for overhead. Therefore, the total number of execution cycles needed for the routine is $33 + 4n$, where n is the filter length. Also, note that data memory and program memory requirements are $5 + 2n$ and $30 + 3n$ words, respectively. For adaptive filters, the filter length is restricted by both execution time and memory. There is obviously more processing to be completed per sample due to the adaptation, and the adaptation

```
       TITL  'ADAPTIVE FILTER'
       DEF   ADPFIR
       DEF   X,Y
*
* THIS 256-TAP ADAPTIVE FIR FILTER USES ON-CHIP MEMORY BLOCK
* BO FOR COEFFICIENTS AND BLOCK B1 FOR DATA SAMPLES. THE
* NEWEST INPUT SHOULD BE IN MEMORY LOCATION X WHEN CALLED.
* THE OUTPUT WILL BE IN MEMORY LOCATION Y WHEN RETURNED.
* ASSUME THAT THE DATA PAGE IS O WHEN THE ROUTINE IS CALLED.
*
COEFFP EQU   >FF00        ; BO PROGRAM MEMORY ADDRESS
COEFFD EQU   >0200        ; BO DATA MEMORY ADDRESS
*
ONE    EQU   >7A          ; CONSTANT ONE           (DP=0)
BETA   EQU   >7B          ; ADAPTATION CONSTANT (DP=0)
ERR    EQU   >7C          ; SIGNAL ERROR           (DP=0)
ERRF   EQU   >7D          ; ERROR FUNCTION      (DP=0)
Y      EQU   >7E          ; FILTER OUTPUT       (DP=0)
X      EQU   >7F          ; NEWEST DATA SAMPLE  (DP=0)
FRSTAP EQU   >0300        ; NEXT NEWEST DATA SAMPLE
LASTAP EQU   >03FF        ; OLDEST DATA SAMPLE
*
* FINITE IMPULSE  RESPONSE (FIR) FILTER.
*
ADPFIR CNFP               ; CONFIGURE BO AS PROGRAM:
       MPYK  0            ; Clear the P register.
       LAC   ONE,14       ; Load output rounding bit.
       LARP  AR3
       LRLK  AR3,LASTAP   ; Point to the oldest sample.
FIR    RPTK  255
       MACD  COEFFP,*-    ; 256-tap FIR filter.
       CNFD               ; CONFIGURE BO AS DATA:
       APAC
       SACH  Y,1          ; Store the filter output.
       NEG
       ADD   X,15         ; Add the newest input.
       SACH  ERR,1        ; err(i) = x(i) - y(i)
*
* LMS ADAPTATION OF FILTER COEFFICIENTS.
*
       LT    ERR
       MPY   BETA
       PAC                ; errf(i) = beta * err(i)
       ADD   ONE,14       ; ROUND THE RESULT.
       SACH  ERRF,1
*
       MAR   *+
       LAC   X            ; INCLUDE NEWEST SAMPLE.
       SACL  *
*
       LRLK  AR2,COEFFD   ; POINT TO THE COEFFICIENTS.
       LRLK  AR3,LASTAP   ; POINT TO THE DATA SAMPLES.
       LT    ERRF
       MPY   *-,AR2       ; P = 2*beta*err(i)*x(i-255)
*
ADAPT  ZALR  *,AR3        ; LOAD ACCH WITH b255(i) & ROUND.
       MPYA  *-,AR2       ; b255(i+1) = b255(i) + P
*                         ; P = 2*beta*err(i)*x(i-254)
       SACH  *+           ; STORE b255(i+1).
*
       ZALR  *,AR3        ; LOAD ACCH WITH b254(i) & ROUND.
       MPYA  *-,AR2       ; b254(i+1) = b254(i) + P
*                         ; P = 2*beta*err(i)*x(i-253)
       SACH  *+           ; STORE b254(i+1).
*
       ZALR  *,AR3        ; LOAD ACCH WITH b253(i) & ROUND.
       MPYA  *-,AR2       ; b253(i+1) = b253(i) + P
*                         ; P = 2*beta*err(i)*x(i-252)
       SACH  *+           ; STORE b253(i+1).
        .
        .
        .
*
       ZALR  *,AR3        ; LOAD ACCH WITH b1(i) & ROUND.
       MPYA  *-,AR2       ; b1(i+1) = b1(i) + P
*                         ; P = 2*beta*err(i)*x(i-0)
       SACH  *+           ; STORE b1(i+1).
*
       ZALR  *,AR3        ; LOAD ACCH WITH b0(i) & ROUND.
       APAC  *-,AR2       ; b0(i+1) = b0(i) + P
       SACH  *+           ; STORE b0(i+1).
*
       RET                ; RETURN TO CALLING ROUTINE.
```

Figure 7. 256-tap adaptive FIR filter routine.

itself dictates that the coefficients be stored in the reconfigurable block of on-chip RAM. Thus, an adaptive filter with no external data memory is limited to 256 taps.

Telecommunications applications. Digital signal processing will be more extensively used in telecommunications as it evolves toward all-digital networks.[17] Below, we discuss several typical uses of the TMS320C25 in telecommunications applications.

Echo cancellation. In echo cancellation, an adaptive FIR filter performs the modeling routine and signal modifications needed to adaptively cancel the echo caused by impedance mismatches in telephone transmission lines. The TMS320C25's large on-chip RAM of 544 words and on-chip ROM of 4K words allow it to execute a 256-tap adaptive filter (32-ms echo cancellation) without external data or program memory.

High-speed modems. For high-speed modems, the TMS320C25 can perform functions such as modulation and demodulation, adaptive equalization, and echo cancellation.[18,19]

Voice coding. Voice-coding techniques such as full-duplex, 32,000-bit-per-second adaptive differential pulse-code modulation (CCITT G.721), CVSD, 16,000-bit-per-second subband coding, and linear predictive coding are frequently used in voice transmission and storage. The speed of the TMS320C25 in performing arithmetic and its normalization and bit-manipulation capabilities enable it to implement these functions, usually within itself (i.e., with no external devices).

Graphics and image processing applications. In these applications, a signal processor's ability to interface with a host processor is important. The TMS320C25 multiprocessor interface enables it to be used in a variety of host/coprocessor configurations. Graphics and image processing applications can use the TMS320C25's large directly addressable external data space and global memory capability to allow graphical images in memory to be shared with a host processor, thus minimizing data transfers. The TMS320C25's indexed indirect addressing modes allow matrices to be processed row by row when matrix multiplication is performed for 3-D image rotation, translation, and scaling.

High-speed control applications. These applications use the TMS320C25's general-purpose features for bit-test and logical operations, timing synchronization, and fast data transfers (10 million 16-bit words per second). They use the TMS320C25 in closed-loop systems for control signal conditioning, filtering, high-speed computing, and multichannel multiplexing. The following examples demonstrate typical control applications.

Disk control. In disk drives, a closed-loop actuation mechanism positions the read/write heads over the disk surface. Accurate positioning requires various signal conditioning tasks to be performed. The TMS320C25 can replace costly bit-slice, custom, and analog solutions in performing such tasks as compensation, filtering, and fine/coarse tuning.

Robotics. The TMS320C25's digital signal processing and bit-manipulation power, coupled with its host interface, allow it to be useful in robotics control. The TMS320C25 can replace both the digital controllers and the analog signal processing hardware a robot needs to communicate to a central host processor, and it can perform the numerically intensive control functions typical of robotic applications.

Instrumentation. Instruments such as spectrum analyzers often require a large data memory space and a processor capable of performing long-length FFTs and generating high-precision functions with minimal external hardware. The TMS320C25 fulfills these requirements.

Numeric processing applications. Numeric and array processing applications benefit from the TMS320C25's performance. The device's high throughput and its multiprocessing and data memory expansion capabilities make it a low-cost, easy-to-use replacement for a typical bit-slice array processor.

Benchmarks. The TMS320C25 has demonstrated impressive performance of benchmarks representing common DSP routines and applications. Table 3 shows this performance.

The TMS320C25 digital signal processor is the newest member of the TMS320 family. It is a pin-compatible, CMOS version of the TMS32020 but offers several enhancements of that device—a 100-ns instruction cycle time, 4K words of on-chip masked ROM, eight auxiliary registers, an eight-level hardware stack, and a double-buffered serial port. It also enhances the TMS32020 instruction set to support adaptive filtering, extended-precision arithmetic, bit-reversed addressing, and faster I/O.

The TMS320C25's multiprocessor capability, large memory spaces, and general-purpose features allow it to be used in a variety of systems, including ones currently employing costly bit-slice processors or custom ICs.

Table 3.
TMS320C25 benchmarks.

DSP ROUTINES/APPLICATIONS	PERFORMANCE
FIR filter tap	100 ns per tap
256-tap FIR filter sample rate	37 kHz
LMS adaptive FIR filter tap	400 ns per tap
256-tap adaptive FIR filter sample rate	9.5 kHz
Biquad filter element	1 μs
Echo canceller	32 ms per single chip (with internal memory)
32,000-bit/s CCITT ADPCM	1 channel full-duplex, single-chip (with internal memory)
16,000-bit/s subband coding	2 channels full-duplex, single-chip (with 0.5K external data memory)
2400-bit/s LPC-10 coding	2 channels full-duplex, single-chip (with 2K external data memory)

References

1. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, N.J., 1975.
2. A. V. Oppenheim, ed., *Applications of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
4. *Digital Signal Processing Applications with the TMS320 Family*, Texas Instruments Inc., 1986.
5. R. Morris, *Digital Signal Processing Software*, DSPS Inc., Ottawa, Ont., 1983.
6. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1975.
7. C. Burrus and T. Parks, *DFT/FFT and Convolution Algorithms*, John Wiley & Sons, New York, 1985.
8. K. McDonough, E. Caudel, S. Magar, and A. Leigh, "Microcomputer with 32-bit Arithmetic Does High-Precision Number Crunching," *Electronics*, Feb. 24, 1982, pp. 105-110.
9. S. Magar, E. Caudel, and A. Leigh, "A Microcomputer with Digital Signal Processing Capability," *Digest of Tech. Papers—1982 IEEE Int'l Solid-State Circuits Conf.*, pp. 32-33 and 284-285.
10. *TMS32010 User's Guide*, Texas Instruments Inc., 1983.
11. *TMS320C25 User's Guide*, Texas Instruments Inc., 1986.
12. *TMS32020 User's Guide*, Texas Instruments Inc., 1985.
13. H. G. Cragon, "The Elements of Single-Chip Microcomputer Architecture," *Computer*, Vol. 13, No. 10, Oct. 1980, pp. 27-41.
14. S. Rosen, "Electronic Computers: A Historical Survey," *Computing Surveys*, Vol. 1, No. 1, Mar. 1969.
15. M. Honig and D. Messerschmitt, *Adaptive Filters*, Kluwer Academic Publishers, Hingham, Mass., 1984.
16. *TMS320 Family Development Support Reference Guide*, Texas Instruments Inc., 1986.
17. M. Bellanger, "New Applications of Digital Signal Processing in Communications," *IEEE ASSP Magazine*, July 1986, pp. 6-11.
18. R. Lucky et al., *Principles of Data Communication*, McGraw-Hill, New York, 1965.
19. P. Van Gerwen et al., "Microprocessor Implementation of High Speed Data Modems," *IEEE Trans. Communications*, Vol. COM-25, 1977, pp. 238-249.

Gene A. Frantz has been Texas Instruments' applications manager for digital signal processing products since 1984. He is also a senior member of the Technical Staff at TI. He joined TI in 1974 as a system design engineer and worked on calculators in TI's Consumer Products Division. In 1976 he was assigned to the Li'l Professor design team. He was next assigned to the Speak & Spell project, where he served as program manager. Since then, he has been involved with every speech-related consumer product developed at TI.

Frantz received a BSEE from the University of Central Florida in 1971, an MSEE from Southern Methodist University, and an MBA from Texas Tech University.

Kun-Shan Lin has been involved in digital signal processing applications in the TI Semiconductor Group since 1984. He is a senior member of the TI Technical Staff. He joined Texas Instruments in 1979 and was assigned to the Consumer Products Division, where he developed speech techniques for learning aids. Prior to joining TI, he was an assistant professor of electrical engineering at Tennessee State University and an adjunct assistant professor of EE at the University of New Mexico. Lin received his PhD from the University of New Mexico in 1976.

Jay Reimer, a member of the TI Technical Staff, handles DSP applications engineering for the TMS320 family of products. He joined TI in 1979 to work with speech products in the company's Consumer Products Division. In 1984, he transferred to the Semiconductor Group to work with digital signal processors. His responsibilities include software development for the TMS320 family and applications assistance for customers using the processors. Reimer received a BS in physics from Fort Hays State University, Kansas, in 1975 and an MS in physics from the University of Kansas in 1977.

Jon Bradley is an applications engineer for the TMS320 family. He joined Texas Instruments in 1976 and has been an applications engineer for most of TI's microprocessor and peripheral products, starting with the TMS9900 family. His responsibilities have included microprocessor system design, digital and analog circuit design, integrated circuit design, test engineering, and programming. Bradley received a BSEE from Worcester Polytechnic Institute, Massachusetts, in 1976.

Questions about the products discussed in this article can be directed to Texas Instruments, Digital Signal Processing Dept.—Mail Station 6400, PO Box 1443, Houston, TX 77001; (713) 879-2320.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 150 Medium 151 Low 152

The Motorola DSP56000 Digital Signal Processor

The DSP56000 brings 10.25-MIPS performance to digital signal processing and retains enough similarities to other Motorola microprocessors to make it easy to learn and program.

Kevin L. Kloker
Motorola, Inc.

The Motorola DSP56000 is a high-performance, user-programmable digital signal processor implemented in high-density, low-power CMOS technology. The first member of a new family of special-purpose microprocessors designed specifically for digital signal processing applications,[1,2] it has a highly parallel architecture and can execute 10.25 million instructions per second. Its instruction set supports the real-time processing of 24-bit data with 56 bits of internal arithmetic precision. Combining a core processor, RAM, ROM, peripheral interfaces, and a memory expansion interface on a single, 88-pin chip, the DSP56000 represents the state of the art in digital signal processor design.

Though the DSP56000 has hardware and software features specifically added to support digital signal processing, it shares enough similarities with other Motorola microprocessors and single-chip microcomputers that it can be used as a very fast microprocessor in any application that requires high-speed calculations or real-time response. For example, it can be used as a controller in applications previously served by bit-slice processors. Providing both single-chip and expanded bus operation, it also fits many applications beyond the capabilities of 8- and 16-bit microcontrollers.

The digital signal processing environment

Digital signal processing is concerned with the real-time processing of digitized analog signals, which are discrete in both amplitude and time.[3] To illustrate the environment of a typical DSP system, I will consider a simple digital filtering application (Figure 1). The system shown is intended to filter an analog signal using digital means. The system starts with an analog input signal $x(t)$, which is converted to a sampled digital signal $x(n)$ by an analog-to-digital, or A/D, converter. As long as the system samples the analog input at a frequency fs that is at least twice the information bandwidth of that input, all information present in the original analog signal is contained in the digital signal. No signal information is lost. However, the quality of the conversion to a discrete amplitude signal introduces quantization noise into the system. The sampled digital signal—that is, its signal-to-quantization-noise ratio, or SQNR—is a function of the A/D converter's accuracy. Choosing the resolution (number of bits) and linearity of the A/D converter is a trade-off between cost and the SQNR requirements of the system.[4]

Once the system has obtained a faithful digital representation of the analog signal, it can filter that representation in the digital signal processor. The digital signal processor stores the current A/D sample and $N-1$ previous samples in a sample shift register. The DSP56000, however, stores this data in a RAM and simulates the shift register function by modifying memory address pointers. The set of N filter coefficients $h(i)$, $i = 0, 1, \ldots, N-1$, must also be stored in a RAM or ROM; these coefficients determine the impulse response and the filter characteristics. A larger N gives a longer impulse response and generally produces filters with sharper roll-off, greater stopband attenuation, and less fre-

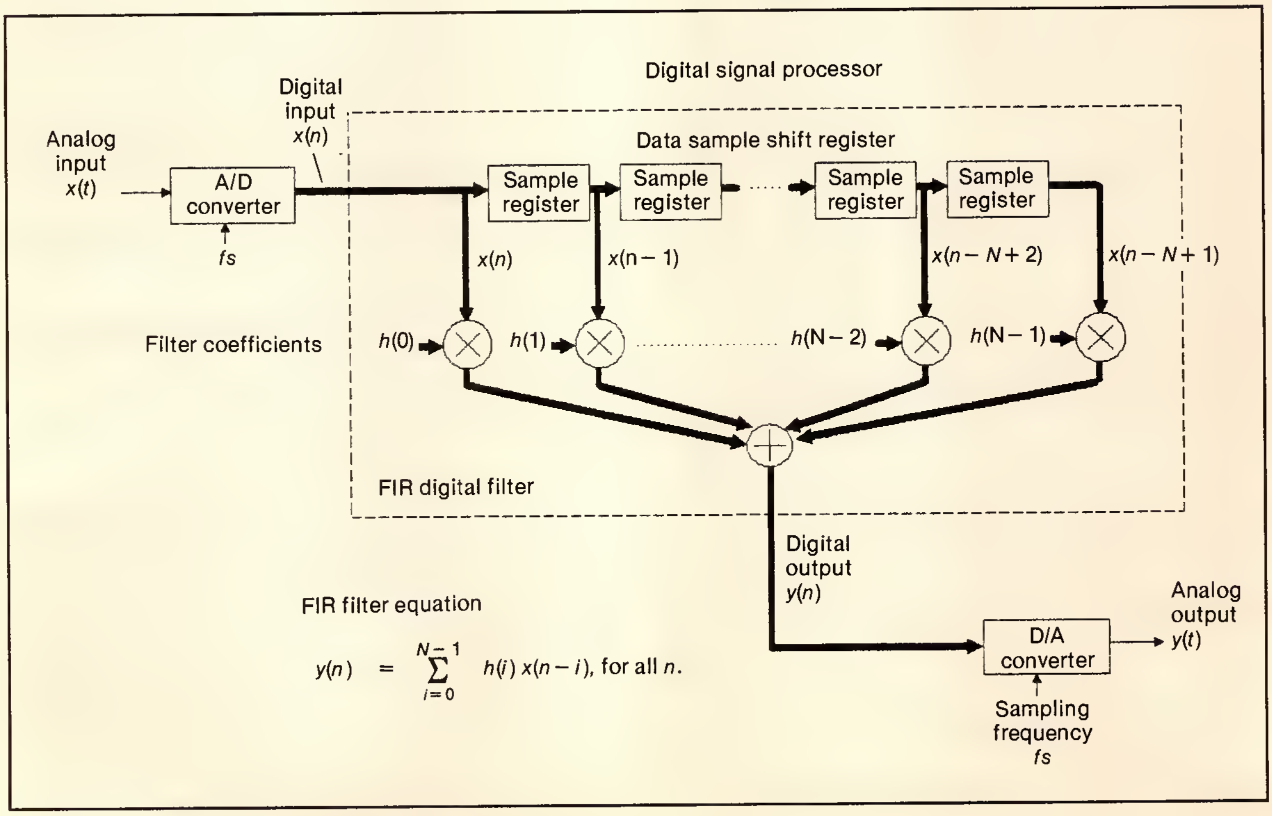
0272-1732/86/1200-0029$01.00 © 1986 IEEE

quency ripple. It does so at the expense of more calculations per output sample and more storage for sample data and coefficients. Since the order N of the filter is finite and no feedback path exists to sustain a nonzero filter output given a zero input, the filter is called an Nth order, finite impulse response, or FIR, digital filter. The FIR filtering operation requires N multiplies and $N-1$ additions to compute an output $y(n)$ each time the input signal is sampled. This operation is the kernel of digital signal processing algorithms. The DSP56000 contains a parallel hardware multiply/accumulate circuit that completes this operation in 97.5 ns. Repeated use of the multiply/accumulate operation produces a sum-of-products result, which is the FIR filter output $y(n)$. This output is converted to an analog signal $y(t)$ by a digital-to-analog, or D/A, converter. The analog output $y(t)$ is now a filtered version of the analog input $x(t)$.

The operation of the system shown in Figure 1 is straightforward. A sample is taken, a filter calculation is performed, and a filter output is sent to the D/A converter. The data samples are then time-shifted (delayed) by one sampling period and the operations are repeated in the next sampling period. A large amount of data must be stored, processed, and time-shifted during each sampling period. In practice, digital signal processing systems can be much more complex than this simple example. However, the example shows what characterizes every digital signal processing environment—multiply/accumulates, time shifts, and input/output in real time.

DSP advantages. The primary advantage of digital signal processing is that it transforms analog functions into digital hardware. It extends the advantages of digital circuitry—high density, precision, stability, and testability—to those parts of systems previously served by analog components. Digital signal processing also transforms analog functions in hardware into digital signal processing algorithms. Traditional analog functions such as filters are replaced by their digital equivalents in software form. Software is inherently flexible and thus is better suited to complex systems. Finally, digital signal processing implements applications that analog signal processing cannot. It makes possible digital speech transmission, storage, and synthesis for communications systems, for example.[5]

DSP limitations. DSP users cite the inability of existing devices to provide high processing speed and large memory sizes as one of the most frequent system limitations. Some DSP applications involve sampling rates of up to 100 MHz and can require hundreds of millions of multiply/accumulates per second. Another problem exists as well. Because DSP is a new discipline, engineers are not familiar with it and require training and experience in it. To apply digital signal processing techniques, they must rethink traditional product development approaches.



$$y(n) = \sum_{i=0}^{N-1} h(i)\, x(n-i), \text{ for all } n.$$

Figure 1. A simple digital filtering system.

DSP56000 features

The DSP56000 incorporates many new features designed to overcome the limitations of earlier DSP devices. It advances the state of the art in three areas—speed, precision, and integration.

Speed. Invariably, DSP users identify processor speed and system performance in a real-time, I/O-intensive environment as the most important factors in their designs. The speed with which a digital signal processor can execute DSP algorithms is critical to the realization of real-time operation. The DSP56000 achieves 10.25 MIPS and 10.25 million multiplies and accumulates per second, record highs for its class of device. It executes many DSP benchmarks two to four times faster than other recently announced DSP devices.[6-8] To achieve this performance, the DSP56000's designers developed a unique multiplier/accumulator ALU and a multiple-bus architecture. They also designed the instruction set and interrupt structure to minimize software overhead in real-time systems.

Precision. The DSP56000 provides the greater data precision needed for advanced digital signal processing applications. Users often ask for more than 16-bit data, since they often employ double-precision arithmetic, data scaling, and overflow checks to maintain 16-bit precision. Yet many users do not need the dynamic range of 32-bit floating-point arithmetic, since the majority of data converters they use are 16 bits or less. Hence, the DSP56000's designers selected a 24-bit fixed-point data format, which provides 24-bit precision without sacrificing system speed or silicon. The 24-bit data word provides 144 dB of external dynamic range, sufficient for most real-world applications. The designers also chose 56-bit accumulators internal to the data ALU to provide 336 dB of internal dynamic range. With 56-bit accumulation, no precision is lost during intermediate calculations. Twenty-four-bit precision provides eight bits of margin against overflow, round-off, and truncation errors when 16-bit data are being processed and eliminates the need for extra processing steps to maintain maximum precision. The larger data size maximizes speed by allowing all calculations to be performed in single-precision fashion.

Integration. A typical DSP system such as that shown in Figure 1 contains two independent data memories (one for data samples and one for coefficients), a separate program memory, and a parallel or serial interface to peripheral devices such as A/D and D/A converters. Anticipating such system needs, the DSP56000 includes five on-chip memories, three on-chip peripherals, and a full-speed memory expansion port. Putting ample memory and peripheral resources on the chip maximizes speed and minimizes system chip count. By integrating the elements of a DSP system in a single CMOS chip, the DSP56000 provides a cost-effective solution to DSP users. Many DSP systems incorporate host microprocessors and additional DSP ICs, memory, and peripherals. By providing both parallel and serial input/output capability in an 88-pin package, the DSP56000 supports such expanded systems.

DSP56000 architecture

A block diagram of the DSP56000 architecture is shown in Figure 2. The architecture consists of a core processor that executes the DSP56000 instruction set and other on-chip resources such as memory and peripherals that provide storage and I/O capability to the core processor. On-chip memory and peripherals are not considered part of the core and may vary from one device family member to another. The chip's pins interface both the core processor and on-chip peripherals to external devices.

Core processor. The core processor consists of three separate execution units—the data ALU, the address ALU, and the program controller—connected by multiple buses. These units operate in a parallel rather than in a pipelined fashion; i.e., each execution unit works on the same instruction at the same time. This is in contrast to heavily pipelined processors, which work on a large number of different instructions at the same time.[9,10] Working in parallel to minimize latency, the three execution units provide all the resources the DSP56000 needs to execute instructions in a single instruction cycle. Here, I define an instruction cycle as two clock cycles; thus, an instruction cycle is 97.5 ns with a 20.5-MHz processor clock. Each execution unit is itself single-cycle and nonpipelined. The architecture of each execution unit is different, being optimized to support its role in instruction execution. Each unit contains a set of registers, arithmetic elements, and executable operations. These operations are register-oriented rather than memory-oriented. Each execution unit operates on its own local registers—source operands are read from registers within the execution unit and modified by arithmetic element operations, and the results are stored in registers within the same execution unit. Data transfers between execution units or between execution units and memory occur in parallel with internal execution unit operations. The single-cycle, register-oriented execution units are key to DSP56000 performance because they do not impose pipelining latency restrictions on the user.

Programming model

The DSP56000 user programming model is shown in Figure 3. It provides numerous register resources that are partitioned into the three execution units of the processor. The instruction set is designed to allow flexible control of these parallel processing resources. Many instructions allow the programmer to keep each execution unit busy, thus enhancing program execution speed. The user can easily program the DSP56000 from the programming model without needing to refer to a complicated hardware block diagram. Providing this capability to the user is a significant achievement, since typical DSP IC programming resembles writing microcode for a bit-slice architecture. With a typical DSP IC, the user is often burdened with difficult programming resulting from the timing complexities of pipelined data paths. The DSP56000 takes a different approach. Because of its nonpipelined execution units, timing com-

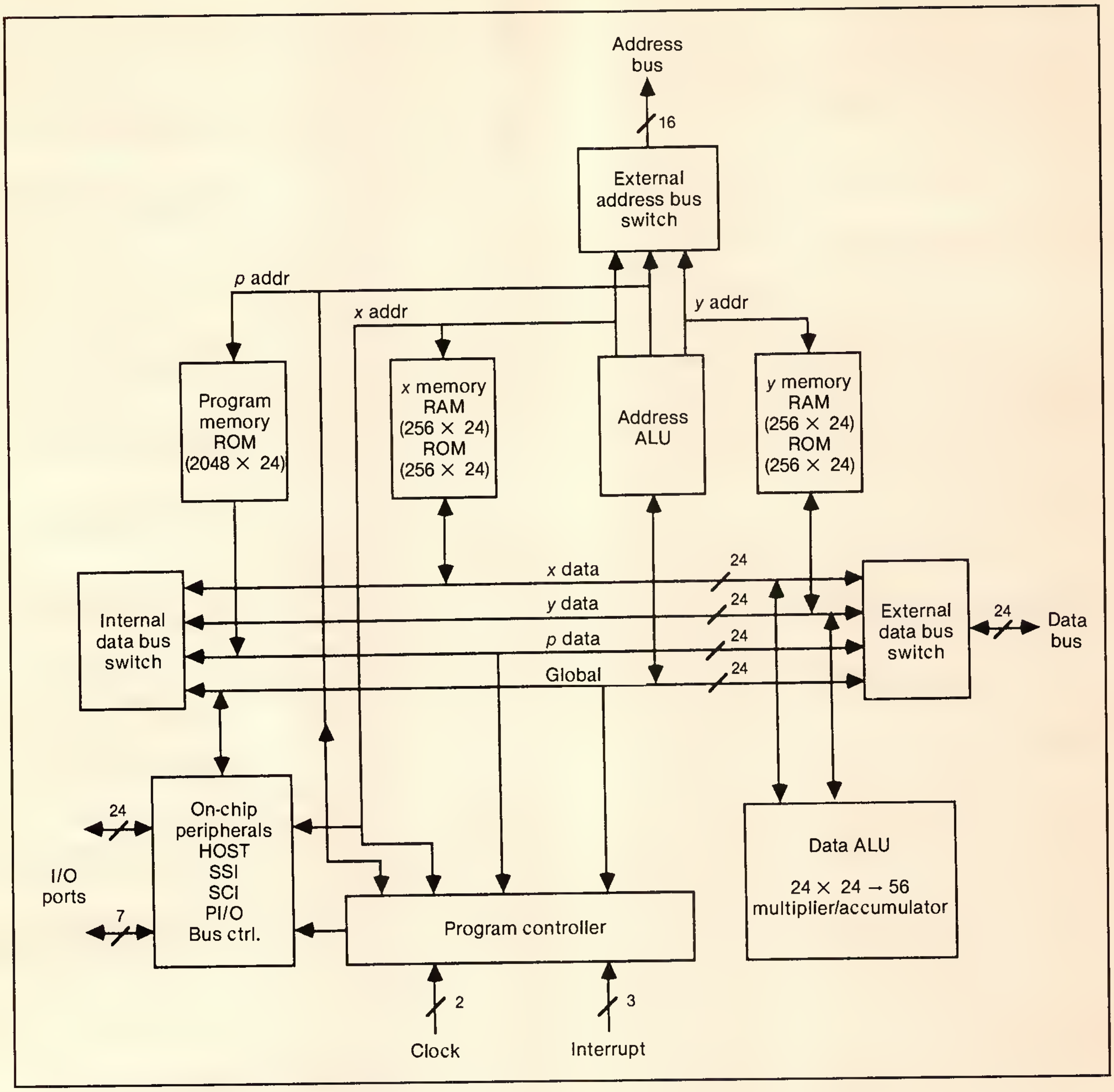

Address bus
16
External address bus switch
p addr
x addr
y addr
Program memory ROM (2048 × 24)
x memory RAM (256 × 24) ROM (256 × 24)
Address ALU
y memory RAM (256 × 24) ROM (256 × 24)
x data
y data
p data
Global
24
24
24
24
Internal data bus switch
External data bus switch
24
Data bus
24
I/O ports
7
On-chip peripherals HOST SSI SCI PI/O Bus ctrl.
Program controller
Data ALU
24 × 24 → 56 multiplier/accumulator
2
Clock
3
Interrupt

Figure 2. DSP56000 block diagram.

plexity is hidden from the user. No microcode is required and a conventional programming model, instruction set, and addressing mode definition can be used. Thus, DSP56000 programming resembles assembly language programming. The regularity of the programming model also makes possible high-level-language programming of the DSP56000.

The status register (SR) format is shown in Figure 4. The condition code register (CCR) portion indicates the results of operations on data operands. The mode register (MR) portion contains information about the system state of the processor.

The instruction set defines three separate memory spaces, *x* data, *y* data, and *p* program, which are each 65,536 locations by 24 bits wide. The total addressing capability is 196,608 24-bit words, or 589,824 bytes. All three memory spaces can be accessed in parallel in the same instruction cycle. The memory maps of these spaces are shown in Figure 5 for the normal expanded operating mode. (One obtains other memory maps by changing the operating mode register, OMR.) Noncore resources (memory and peripherals) are memory-mapped into these memory spaces to provide a clean hardware and software interface with the core processor.

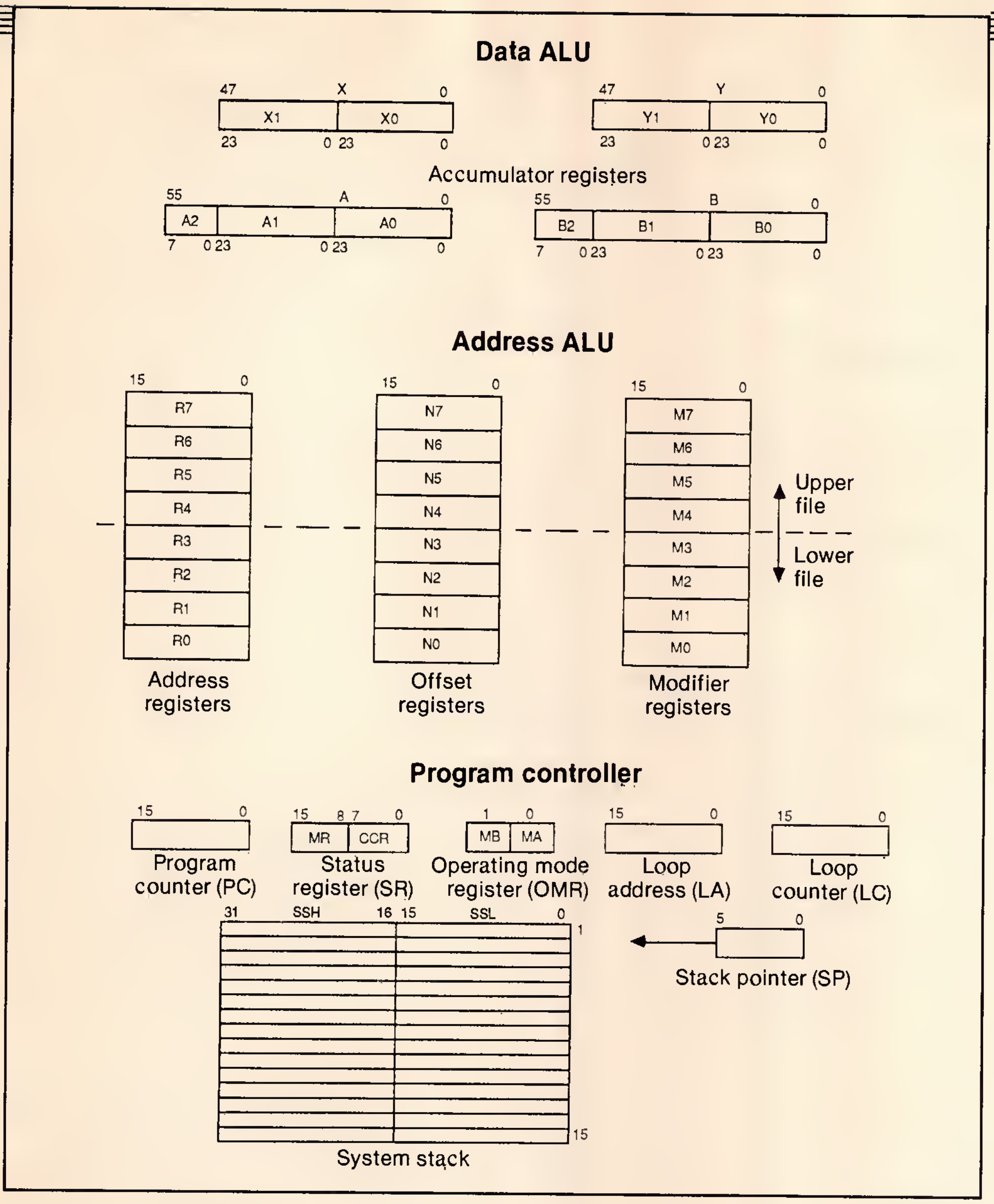
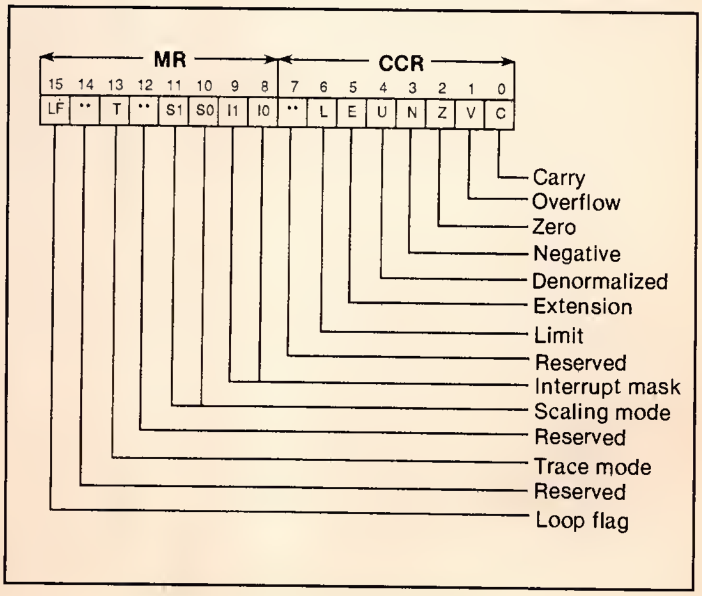


Figure 3. DSP56000 user programming model.

Figure 4. DSP56000 status register.



Data ALU

The data ALU execution unit performs arithmetic and logical operations on data operands. It consists of 10 local registers, two shifter/limiter blocks, an accumulator shifter, and a multifunction multiplier/accumulator (MAC) ALU that has two 56-bit inputs and one 56-bit output.[11] The two 56-bit inputs can be used for operations such as addition, subtraction, and comparison on 24-, 48-, or 56-bit numbers. During multiply and multiply/accumulate operations, one of the 56-bit inputs serves as the accumulator input while the other 56-bit input is reconfigured as a 24-bit multiplicand and 24-bit multiplier input. The MAC ALU contains a 24 × 24-bit parallel hardware multiplier/accumulator circuit that provides 56-bit accumulation. The MAC ALU is not pipelined and performs all operations in a single 97.5-ns instruction cycle. This is in contrast to common two-stage pipelined architectures employing a multiplier separated from an adder by a product pipeline register. The product pipeline register adds an extra cycle of delay before the

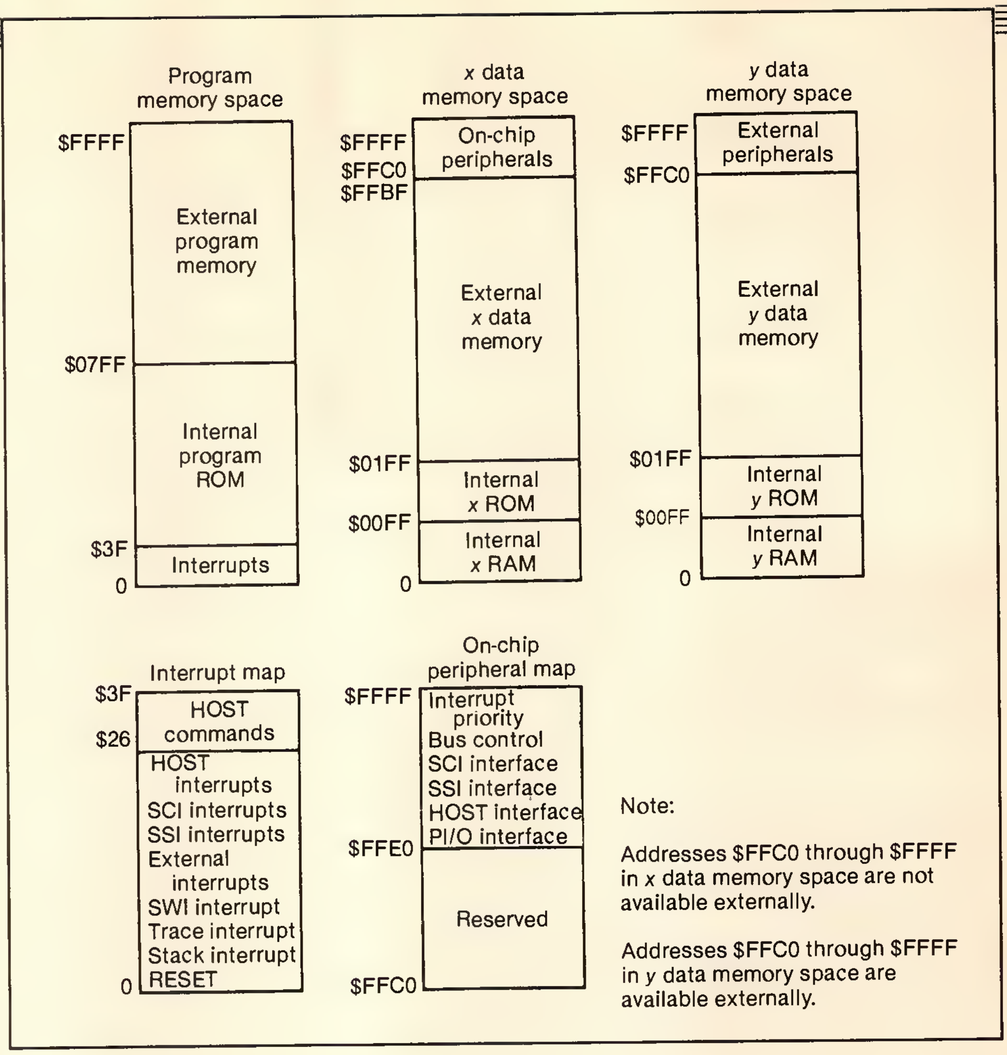


Figure 5. DSP56000 memory maps.

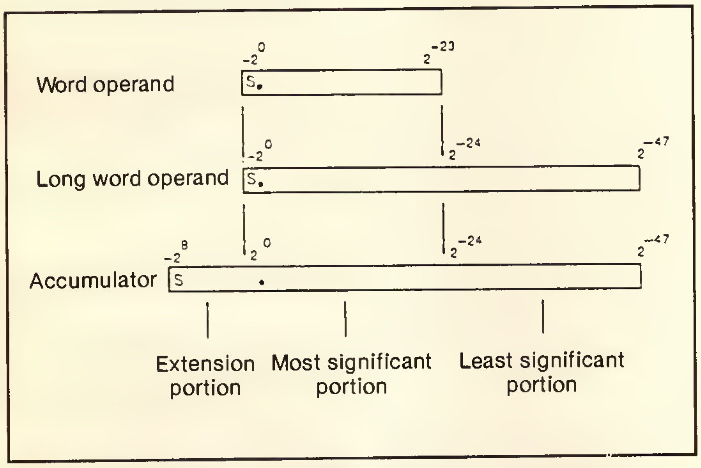


Figure 6. Bit weight and alignment of fractional data.

multiplier output is available. Since the MAC ALU has no product pipeline register, it exhibits no delay and executes many algorithms faster.

The data ALU has four 24-bit general-purpose input registers and six special output registers organized as two 56-bit (8 + 24 + 24-bit) accumulators. The input registers can be read or written as 24- or 48-bit data. The two 56-bit accumulator registers can be accessed in numerous ways as 24-, 48-, or 56-bit data. The MAC ALU output is always stored in an accumulator register. Data ALU operations support several different levels of precision, as indicated in Figure 6. Word operands are 24 bits long, long word operands are 48 bits long, and accumulator operands are 56 bits long. The MAC ALU supports the signed, two's-complement fractional data representation commonly used in digital signal processing algorithms. Fractional data, d, has a range of $-1.0 \leq d < +1.0$ and is identical to signed integer data except for the placement of the binary radix point. In practice, fractional and integer arithmetic are equivalent for all operations except multiply and divide. An integer multiplication product can be formed by a one-bit right shift of a fractional product, for example.

The data ALU provides the standard microprocessor set of logical and shifting operations to support a variety of algorithms. Logical operations are performed in the MAC ALU. A 56-bit accumulator shifter is included on one of the 56-bit MAC ALU inputs for one-bit left or right shifts. The data ALU does not contain a barrel shifter, but the MAC ALU can be used for multibit shifting operations. By multiplying the 24-bit data by a constant or variable, the

MAC ALU can perform left or right shifts of from 1 to 23 bits in a single instruction. The left-shifted result is available in the least significant portion of the accumulator and the right-shifted result is available in the most significant portion of the accumulator. This method is efficiently used by the instruction set to provide data normalization and denormalization as well as bit-field insertion and extraction.

Accumulator extension. The 56-bit accumulators store a complete 48-bit multiplication product plus eight bits of integer data called an accumulator extension. The extension portion of the 56-bit accumulator provides eight bits of protection against overflow during intermediate calculations and allows at least 256 repetitive multiply/accumulate operations before an overflow can occur. The extension bits eliminate the need to scale down the input data to avoid accumulator overflows arising from word growth in repetitive calculations. This is true because the fractional product, p, has a range of $-1.0 \leq p < +1.0$ and the 56-bit accumulator, a, has a range of $-256.0 \leq a < +256.0$.

When word or long word data are written to an accumulator register, the programmer may sign-extend its extension portion and zero its least significant portion to form a valid 56-bit signed number. When 56-bit accumulator data are read out of the data ALU, the programmer may post-process the data by enabling two special shifter/limiter circuits located between the accumulator registers and the data ALU outputs. In operation, a copy of the accumulator data is shifted left or right one bit if it has been enabled by the scaling mode bits in the status register. This allows block floating-point algorithms to be implemented for fast Fourier transforms and matrix manipulation. After shifting, the data are passed through an overflow protection circuit called a limiter. If the accumulator data can be stored in the destination without overflow, the limiter is disabled and the data are not modified. If the accumulator data cannot be stored in the destination without overflow (because the extension portion of the accumulator is in use), the limiter is enabled and substitutes the maximum data value (having the same sign) that the destination can store without overflow. This technique, called saturation arithmetic, minimizes the overflow error by avoiding the sign change usually associated with binary overflow.

Rounding. For many DSP algorithms, single-precision (24-bit) data ALU results are needed to minimize data storage requirements or to provide data for subsequent use as a multiplier input. Rounding the least significant portion of the accumulator into the most significant portion is better than truncation for maintaining maximum precision and avoiding introduction of a negative bias error. The DSP56000 provides "round to nearest even," or RN, rounding, which is performed during multiply and round (MPYR), multiply/accumulate and round (MACR), and round (RND) instructions.

Table 1.
DSP56000 addressing modes.

Addressing mode	Address modifier	Assembler syntax
Register direct	No	Any register name
Address register indirect		
No update	Yes	(R*n*)
Postincrement by 1	Yes	(R*n*)+
Postdecrement by 1	Yes	(R*n*)−
Postincrement by offset N*n*	Yes	(R*n*)+N*n*
Postdecrement by offset N*n*	Yes	(R*n*)−N*n*
Predecrement by 1	Yes	−(R*n*)
Index by offset N*n* (R*n* and N*n* are unchanged)	Yes	(R*n*+N*n*)
Special		
Immediate data (24-bit)	No	#expr
Absolute address (16-bit)	No	expr
Immediate short data (8-, 12-bit)	No	#expr
Short jump address (12-bit)	No	expr
Absolute short address (6-bit)	No	expr
I/O short address (6-bit)	No	expr

n = register number 0 to 7
expr = any valid assembler expression

Address ALU

The address ALU execution unit calculates addresses to locate data operands in memory. Two multiply/accumulate input operands are typically required by the data ALU at each instruction cycle. Data ALU results must be stored at a less frequent rate. The address ALU provides a flexible addressing capability through a large address register set, 14 addressing modes, and three types of address arithmetic. It consists of twenty-four 16-bit address registers, two 16-bit address arithmetic units, and an address output multiplexer. The address ALU can provide two independent memory addresses at each instruction cycle and update them with two address register indirect addressing modes. A summary of the addressing modes and their assembler syntax is given in Table 1.

The 24 address registers are organized into three sets of eight registers. The eight address registers R*n* (n = 0 to 7) are used as address pointers to locate data operands in memory. The eight offset registers N*n* are used as optional offset values to update the address registers. The offset registers may contain signed or unsigned data. The eight

Table 2.
Address modifiers.

Modifier register M*n* value	Address update arithmetic for address register R*n*
0	Reverse-carry (bit-reversed)
1	Modulo 2
2	Modulo 3
.	.
.	Modulo (M*n* + 1)
.	.
32766	Modulo 32767
32767	Modulo 32768
.	
.	Reserved
.	
65535	Linear (modulo 65536)

modifier registers M*n* select the type of address arithmetic to be performed when an address register R*n* is to be updated. As shown in Table 2, the contents of the modifier registers are encoded to select the type of address arithmetic—linear, modulo, or reverse carry. The type of address arithmetic defines the type of data structure (array, sample shift register, or queue) being accessed in memory. Each address register R*n* is assigned an offset register N*n* and a modifier register M*n* having the same register number for use in address calculations. For example, the address calculation (R0) + N0 postincrements the contents of address register R0 by the contents of offset register N0 using the type of address arithmetic specified by the contents of modifier register M0.

The address ALU output multiplexer allows any address register R*n* to be used as a pointer to any memory space. This must be done so that pointers will not be duplicated in multiple registers when multiple memory spaces are accessed. For example, complex data pairs are typically stored in two data memory spaces (real part in *x* memory and imaginary part in *y* memory) at the same address. Efficient access to complex data pairs requires each pointer to be able to access both the *x* and *y* data memory spaces.

Address modifiers. During an address calculation, a set of three registers R*n*, N*n*, and M*n* are accessed by the appropriate address arithmetic unit. The contents of the selected modifier register are decoded by the address arithmetic unit so it can determine the type of address arithmetic it should perform on the selected address pointer R*n*. To understand the role of the address modifiers in creating data structures in memory, consider the examplcs of eight-bit address arithmetic shown in Figure 7.

The linear address modifier example is identical to conventional microprocessor address calculations, in which address updating is performed by a conventional adder. The example shows the postincrement by the offset N*n* addressing mode, where the offset register contains the value 5. Linear addressing is most useful for addressing arrays of data.

The reverse-carry address modifier example is performed through the propagation of the adder carry in the reverse direction, i.e., from the most significant bit to the least significant bit of the adder. A characteristic of typical fast Fourier transform algorithms is that the data and coefficients may be stored in a nonsequential order called bit-reversed order. Reverse-carry addressing can calculate bit-reversed addresses for sequential access of FFT data and coefficients. For a 2^k-point FFT, a postincrement by 2^{k-1} using reverse-carry address arithmetic will generate the bit-reversed address sequence. The example shown generates the bit-reversed address sequence for a 16-point FFT buffer starting at address 64. Reverse-carry addressing is equivalent to doing bit-reversed addressing with simpler hardware.

The modulo address modifier example creates a circular (modulo) address region in memory with a lower boundary and an upper boundary. Modulo arithmetic keeps the address register pointing to a location within the modulo region by automatic wraparound if the pointer increments or decrements out of the modulo region. The modulo size—i.e., the length of the modulo region—is specified by the contents of the modifier register plus one. (The size is 20 in the example.) The modulo size can be any number from 2 to 32768. The lower-boundary address and upper-boundary address of the modulo region need not be directly specified, since the modulo size implicitly defines all possible modulo region boundaries. The lower-boundary address must have as many least significant zeroes as the modifier value has significant bits (five bits for M*n* = 19). For example, a modulo region of size 20 (M*n* = 19) can have a lower-boundary address at any integer multiple of 32 (0, 32, 64, 96, 128, 160, and so on). The upper-boundary address is the lower-boundary address plus the modulo size minus one. One of the possible modulo regions is selected implicitly by loading the address register R*n* pointing to a location within a valid modulo region. (R0 = 75 selects the modulo region from 64 to 83 in the example.) Modulo address arithmetic simulates a shift register in memory by simply updating an address pointer and eliminates the need to move data to perform time-shift functions. Modulo address modifiers can create FIFO queues, delay lines, and sample shift registers in memory. They are also useful for interpolating and decimating filters and for generating periodic waveforms.

Program controller

The program controller execution unit performs instruction flow control, instruction decoding, and exception processing. It consists of a program address generator, or PAG, an instruction decoder, an interrupt controller, and a bus controller. The PAG is nonpipelined and calculates a new

Linear address modifier

M0 = 255 = 1111 1111 for linear addressing with R0.

Original registers: N0 = 5, R0 = 75 = 0100 1011

Postincrement by offset N0: R0 = 80 = 0101 0000

Postincrement by offset N0: R0 = 85 = 0101 0101

Postincrement by offset N0: R0 = 90 = 0101 1010

Reverse-carry address modifier

M0 = 0 = 0000 0000 for reverse-carry addressing with R0.

Original registers: N0 = 8, R0 = 64 = 0100 0000

Postincrement by offset N0: R0 = 72 = 0100 1000

Postincrement by offset N0: R0 = 68 = 0100 0100

Postincrement by offset N0: R0 = 76 = 0100 1100

Modulo address modifier

M0 = 19 = 0001 0011 for modulo 20 addressing with R0.

Original registers: N0 = 5, R0 = 75 = 0100 1011

Postincrement by offset N0: R0 = 80 = 0101 0000

Postincrement by offset N0: R0 = 65 = 0100 0001

Postincrement by offset N0: R0 = 70 = 0100 0110

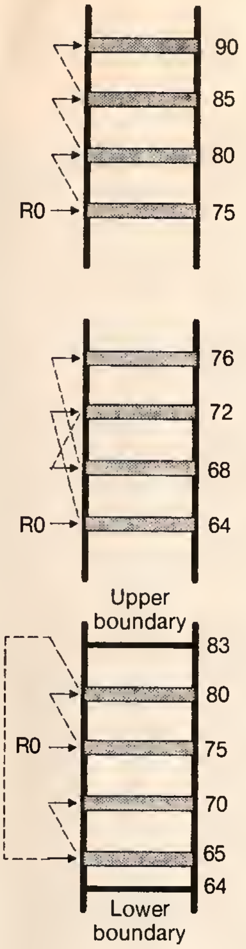


Figure 7. DSP56000 address arithmetic (8-bit examples).

instruction address every instruction cycle. Instruction prefetching is used to form a two-stage instruction pipeline having the basic timing shown in Table 3. One instruction cycle is used for the fetch operation, one for the decode operation, and one for the execute operation. During normal instruction execution, the PAG is responsible for fetching the instruction word two locations ahead of the currently executing instruction. The instruction fetch and decode operations overlap with instruction execution and take no execution time, with the exception of change-of-flow instructions and possibly PAG bus accesses. Change-of-flow instructions must fetch two instruction words to refill the instruction pipe. The PAG may not have immediate access to on-chip program memory if the currently executing instruction is accessing a data operand in on-chip program memory. The PAG may also introduce wait states during external instruction fetches because of slow off-chip program memory or because the currently executing instruction is using the memory expansion port. During a resource conflict, instruction fetches have lower priority than data accesses, since the currently executing instruction has the highest priority for use of chip resources.

Hardware DO loops. Many DSP programs spend 90 percent of the time executing in 10 percent of the program code. Digital filtering routines usually consist of a small code kernel that is executed many times. The DSP56000 provides hardware DO loop control to replace the software "decrement counter and branch" instruction normally associated with DO loops. Straight-line coding is not needed to maximize speed since compact, looped code runs at the same speed as straight-line code. When executing DO loops, the DSP56000 eliminates the usual change-of-flow overhead by modifying the PAG instruction fetch sequence. It initiates a hardware DO loop by executing the DO instruction. The hardware DO loop mechanism stores the loop count (LC), loop starting address, and loop ending address (LA) in special registers and processes them in parallel with the executing program. Inside the DO loop, the instruction fetch address is compared to the loop ending address. When the end of the loop is detected, the loop count is tested for one. If the loop count is not one, it is decremented and the instruction word at the loop starting address is fetched. If the loop count is one, the DO loop execution is complete and normal sequential instruction fetches resume. The saving and restoring of the previous hardware DO loop registers on a stack makes it possible to nest hardware DO loops with minimal overhead. A separate system stack is used to minimize overhead during nested hardware DO loops and multilevel interrupts. This hardware stack is 32 bits wide and 15 locations deep. The double width allows two registers to be transferred to/from the system stack every instruction cycle. One can extend the system stack to any depth by moving the stack data to/from memory using software stacking techniques.

Exception processing. Exception processing in a DSP environment is primarily associated with the transfer of data

Table 3.
DSP56000 instruction timing.

Normal instruction timing

Instruction cycle	1	2	3	4	5	6	7	8	9	10
Fetch	n3	n4	n5	n6	n7	n8	n9	n10	n11	n12
Decode	n2	n3	n4	n5	n6	n7	n8	n9	n10	n11
Execute	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10

n = normal instruction word

Fast interrupt instruction timing

Instruction cycle	1	2	3	4	5	6	7	8	9	10
Fetch	n3	n4	iv1	iv2	n5	n6	n7	n8	n9	n10
Decode	n2	n3	n4	iv1	iv2	n5	n6	n7	n8	n9
Execute	n1	n2	n3	n4	iv1	iv2	n5	n6	n7	n8

n = normal instruction word
iv = interrupt vector instruction word (no change of flow)

Long interrupt instruction timing

Instruction cycle	1	2	3	4	5	6	7	8	9	10
Fetch	n3	n4	iv1	iv2	i1	i2	i3	n5	n6	n7
Decode	n2	n3	n4	JSR	—	i1	RTI	—	n5	n6
Execute	n1	n2	n3	n4	JSR	—	i1	RTI	—	n5

n = normal instruction word
iv = interrupt vector instruction word (change of flow)
JSR = jump to interrupt service routine
i = interrupt service routine instruction word
RTI = return from interrupt

between processor memory or registers and a peripheral device. When an interrupt occurs, a limited context switch must be performed with minimum overhead. Saving all of the machine state is too time consuming and usually not necessary. The DSP56000 provides a sophisticated interrupt structure which reduces the timing overhead associated with servicing interrupts. Each peripheral device and external interrupt pin may be programmed to one of three interrupt priority levels (IPLs) so that time-critical interrupts are always serviced first. When more than one interrupt is pending within the same IPL, a fixed priority table determines the secondary priority level within that IPL. Most on-chip peripherals have separate interrupt vectors for each interrupting condition so the cause of the interrupt will be known before the interrupt service routine is entered. There is no need to poll devices or test status bits to determine the interrupting condition. An interrupt vector may be error-free or not error-free. If it is error-free, no error conditions are associated with the interrupt request, and a data transfer can service the request immediately without checking error flags. If the interrupt vector is not error-free, error conditions are associated with the interrupt request, and the interrupt service routine must first check the error flags. As can be seen from Table 4, 18 out of 32 possible interrupt vectors are used. The two external interrupt pins, IRQA and IRQB, may be programmed as level-sensitive or negative-edge triggered.

The timing of the interrupt controller is shown in Table 3. First, the interrupt controller synchronizes and prioritizes pending interrupts to determine the highest-priority, unmasked interrupt request. Second, the instruction fetch stream is temporarily redirected to fetch two interrupt instruction words (iv1 and iv2) at the interrupt vector addresses. The two interrupt instruction words at the interrupt vector addresses are fetched into the instruction pipeline without waiting for the current instruction to finish execution. The program counter is held constant since the interrupt controller provides the two interrupt vector addresses. Finally, normal instruction fetches (*n*5, *n*6, and so on, according to the program counter's contents) resume immediately after the two interrupt instruction words have been fetched. The usual interrupt vector change-of-flow is avoided and the extra cycles required to empty the instruction pipeline and refill it are eliminated. However, if instruction word *n*4 is the first word of a two-word instruction, the execution of *n*4 is aborted and *n*4 is refetched in place of *n*5. Since most instructions are one word long, this occurs infrequently. The two interrupt instruction words iv1 and iv2 are decoded and executed. They may be two single-word instructions or one two-word instruction.

Fast interrupts. If execution of the two interrupt instruction words does not cause a change of flow, the interrupt routine is called a fast interrupt. In a fast interrupt, normal instruction execution continues without delay following the execution of the two interrupt instruction words iv1 and iv2. Fast interrupts do not save the machine state, so instructions that modify the machine state should not be used. Fast interrupts do not require a return from interrupt (RTI) instruction, since no context switch is performed. Although any non-change-of-flow instruction can be used, a special move peripheral (MOVEP) instruction has been provided to support fast interrupts with a memory-to-memory data transfer between memory-mapped peripheral devices and any memory space. Fast interrupts can process up to 1.7 million interrupts (or 10.25 million bytes) per second, yet they consume only 33 percent of the total execution time. This performance level rivals that of dedicated direct memory access hardware. Fast interrupts are a software alternative to DMA that provides several advantages. Unlike DMA, fast interrupts can service peripheral devices with the flexibility that only software can offer. Fast interrupts can use any addressing mode with linear, modulo, or reverse-carry address arithmetic. They can service on-chip as well as off-chip peripherals and memory using minimal hardware. Fast interrupts can also handle peripheral errors by vectoring to different interrupt service routines and can support termination conditions other than the traditional word count.

Table 4.
Interrupt sources.

Interrupt vector starting address	Error-free status	Interrupting condition
$0000	—	Hardware RESET
$0002	No	Stack error
$0004	Yes	Trace
$0006	Yes	Software interrupt (SWI)
$0008	Yes	External interrupt IRQA
$000A	Yes	External interrupt IRQB
$000C	Yes	SSI receive data
$000E	No	SSI receive data with exception
$0010	Yes	SSI transmit data
$0012	No	SSI transmit data with exception
$0014	Yes	SCI receive data
$0016	No	SCI receive data with exception
$0018	Yes	SCI transmit data
$001A	Yes	SCI idle line
$001C	Yes	SCI timer
$001E	—	Reserved for hardware development
$0020	Yes	HOST receive data
$0022	Yes	HOST transmit data
$0024	Yes	HOST command (default)
$0026	Yes	Available for HOST command (AHC)
$0028	Yes	AHC
.	.	.
.	.	.
$003E	Yes	AHC

Long interrupts. If either of the two interrupt instruction words is a jump to subroutine (JSR) instruction, the interrupt routine is called a long interrupt. Programming a JSR instruction at the interrupt vector addresses iv1 or iv2 causes a long interrupt routine. When the JSR instruction is decoded, the DSP56000 performs a context switch by saving the current program counter and status register on the stack and updating the interrupt mask in the status register. The program counter is loaded with the JSR destination address and the long interrupt routine (i1, i2, and so on) begins execution. If he desires it, the programmer can save more of the machine state by using software stacking operations. The long interrupt routine is terminated by the usual RTI instruction (located at i2 in this example). As shown in Table 3, this mechanism allows long interrupts to be vectored via the JSR destination address with a minimum of timing overhead. This change of flow is as fast as conventional vectored interrupts and is a natural extension of the fast interrupt mechanism.

Multiple-bus architecture

The internal multiple-bus architecture of the DSP56000 consists of four data buses and three address buses connecting the various resources on the chip. Support of the fast multiplier/accumulator requires two multiplier input operands to be transferred between memory and the data

Table 5.
DSP56000 instruction set.

Arithmetic instructions	
ABS	Absolute value
ADC	Add with carry
ADD	Add
ADDL	Shift left then add
ADDR	Shift right then add
ASL	Arithmetic shift left
ASR	Arithmetic shift right
CLR	Clear
CMP	Compare
CMPM	Compare magnitude
DIV	Divide iteration
MAC	Multiply/accumulate
MACR	Multiply/accumulate and round
MPY	Multiply
MPYR	Multiply and round
NEG	Negate
NORM	Normalize iteration
RND	Round
SBC	Subtract with carry
SUB	Subtract
SUBL	Shift left then subtract
SUBR	Shift right then subtract
Tcc	Transfer conditionally
TFR	Transfer
TST	Test

Logical instructions	
AND	Logical AND
ANDI	AND immediate control register
EOR	Logical exclusive OR
LSL	Logical shift left
LSR	Logical shift right
NOT	Complement
OR	Logical inclusive OR
ORI	OR immediate control register
ROL	Rotate left
ROR	Rotate right

Bit manipulation instructions	
BCLR	Bit test and clear
BSET	Bit test and set
BCHG	Bit test and change
BTST	Bit test on memory
JCLR	Jump if bit clear
JSET	Jump if bit set
JSCLR	Jump to subroutine if bit clear
JSSET	Jump to subroutine if bit set

Program control instructions	
Jcc	Jump conditionally
JMP	Jump
JScc	Jump to subroutine conditionally
JSR	Jump to subroutine
NOP	No operation
REP	Repeat next instruction
RESET	Reset on-chip peripheral devices
RTI	Return from interrupt
RTS	Return from subroutine
STOP	Stop processing
SWI	Software interrupt
WAIT	Wait for interrupt

Loop instructions	
DO	Start hardware loop
ENDDO	Exit from hardware loop

Move instructions	
LUA	Load updated address
MOVE	Move data
MOVEC	Move control register
MOVEM	Move program memory
MOVEP	Move peripheral data

ALU upon each instruction cycle. Instruction fetches must also be done at the same rate. This translates to three data transfers (two data and one instruction) every instruction cycle. The bus structure supports general register-to-register, register-to-memory, and memory-to-register data movement and can transfer up to three 24-bit words at the same time. The resources connected to a bus define its primary data-transport function (see Figure 2 again). One address and data bus is associated with each of the three memory spaces (*x* data, *y* data, and *p* program), whereas a fourth data bus, called the global data bus, is shared by all three memory spaces. The role of the global data bus is to physically extend the *x*, *y*, and *p* data buses so they can be connected to remote chip resources while keeping these buses as short as possible. Keeping the buses local enhances the speed of the device.

Internal bus switch. Communication between the buses in the multiple-bus structure takes place in an internal bus switch. The internal bus switch is similar to a switch matrix and can connect any two internal buses without adding any pipeline delays. This flexibility allows a general data move capability for easier programming. Since the internal data bus switch can access each memory space, the bit manipulation unit is physically located in this block.

Table 6.
Parallel move operations.

Parallel move operation	Example of assembler syntax
No parallel move	ADD X0,A
Register → register	ADD X0,A Y1,R0
Address register update	ADD X0,A (R1) – N1
Immediate short data → reg.	ADD X0,A #183,R4
Immediate data → register	ADD X0,A #$F97B4A,B
Immediate data → register plus register → register	ADD X0,A #$123456,X1 B,Y1
Absolute short address ↔ reg.	ADD X0,A Y:$3A,R4
Absolute address ↔ register	ADD X0,A A,X:$FFE3
Absolute address ↔ register plus register → register	ADD X0,A A,X1 Y0,Y:$3F80
x memory ↔ register	ADD X0,A X0,X:(R5) +
x memory ↔ register plus register → register	ADD X0,A X:(R0) – ,X0 A,Y0
y memory ↔ register	ADD X0,A Y:(R0) + ,R7
y memory ↔ register plus register → register	ADD X0,A B,X0 A,Y:(R0) + N0
x memory ↔ register plus *y* memory ↔ register	ADD X0,A X1,X:(R3) + Y:(R6) – ,B
L long memory ↔ register	ADD X0,A AB,L:(R2) +

Here, "source,destination" assembler format is used and "ADD X0,A" is a sample opcode and operand.

DSP56000 instruction set

The DSP56000 has an easy to learn, microprocessor-style instruction set that is efficient for many different algorithms. Its instruction set has some characteristics of a reduced-instruction-set computer because of its register-based (load/store) orientation and because most of the instructions are executed in a single cycle. However, enough complex-instruction-set computer functionality is built into each instruction that the DSP56000 achieves the high performance needed for small code loops typically consisting of only two to five instructions. Lack of data pipelining contributes to the ease of programming and eliminates any architectural bias against an algorithm. During development of the DSP56000, its designers used a set of 15 common DSP benchmarks to test the speed and coding efficiency of the instruction set whenever they made changes to it. These benchmarks included digital filters and fast Fourier transforms for real and complex data. (The benchmarks are listed in an appendix to the DSP56000 manual.[12]) The designers used other benchmarks to measure performance for two-dimensional problems such as matrix manipulation and image processing. Many algorithms achieve high performance on the general-purpose DSP56000. A list of DSP56000 opcodes is shown in Table 5.

In addition to the usual set of microprocessor opcodes, the DSP56000 instruction set provides a powerful set of multiply and multiply/accumulate instructions with options for rounding and positive or negative product accumulation. Other opcode additions include absolute value, shift left or right then add or subtract, compare magnitude, normalize, round, and transfer conditional instructions. When used after a compare or compare magnitude instruction, the transfer conditional (Tcc) instruction can perform maximum value, minimum value, maximum absolute value, minimum absolute value, and other functions.

Parallel move operations. Most arithmetic and logical instructions can specify up to two data transfers in the same instruction. These data transfers are called parallel move operations and are executed in one instruction cycle in parallel with the instruction opcode. This allows two or three conventional instructions to be combined into one parallel instruction, with corresponding gains in speed and coding efficiency. Parallel move operations allow the register-based execution units to be kept busy by concurrently preloading new input operands and storing previous results. They also provide concurrent communications between execution units. Used with the local registers in each execution unit, the parallel move operations provide "software pipelining" controlled by the user. Thus, the user can adapt the DSP56000's parallel architecture to his application. The parallel move operations are shown in Table 6. The assembler syntax samples illustrate the different parallel move operations that can be specified with the same ADD X0,A instruction. Note that the same register may be a source operand more than once in the same instruction.

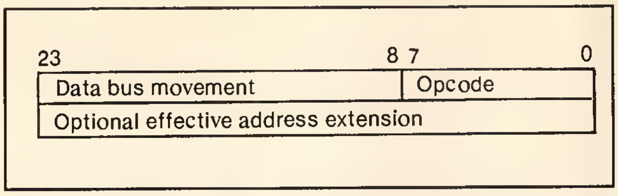


Figure 8. DSP56000 instruction encoding format.

However, no register may be specified as a destination operand more than once in the same instruction.

The general instruction encoding format is shown in Figure 8. All instructions are one or two 24-bit words in length. The first word generally contains an 8-bit opcode field and a 16-bit data bus movement field. The opcode field includes the instruction opcode with its source and destination register operands. The data bus movement field specifies source and destination operands for parallel move operations over the *x* data and *y* data bus. For address register indirect addressing modes, the data bus movement field specifies up to two address registers and associated addressing modes. As shown earlier in Table I, the DSP56000 provides a set of 14 addressing modes to minimize address generation overhead. Both register direct and address register indirect (pointer) addressing modes are available as one-word instructions. Absolute short, I/O short, and short jump addresses and immediate short data addressing modes are also provided, being encoded into the data bus movement field for fast, one-word instructions. Absolute address and immediate data addressing modes are encoded in the second optional effective address extension word to provide full 16-bit addresses and 24-bit data.

On-chip resources

The DSP56000 provides a large set of on-chip memory and I/O peripheral resources to support the core processor. The on-chip memories include two 256 × 24-bit data RAMs, two 256 × 24-bit data ROMs and one 2048 × 24-bit program ROM. Data or program code can be moved from any memory space to another, whether on-chip or off-chip. Microcomputer-style I/O capability is provided by three on-chip peripherals—the parallel host MPU/DMA interface (host), the asynchronous serial communications interface (SCI), and the synchronous serial interface (SSI)—on 24 programmable, general-purpose port pins. Noncore resources are supported by standard move and bit manipulation instructions. This avoids irregular I/O instructions and will allow Motorola to easily change noncore resources in future DSP56000 family members without affecting the instruction set.

Memory expansion port. The DSP56000 chip pinout is shown in Figure 9. Memory expansion off-chip is provided by a memory expansion interface called Port A. External

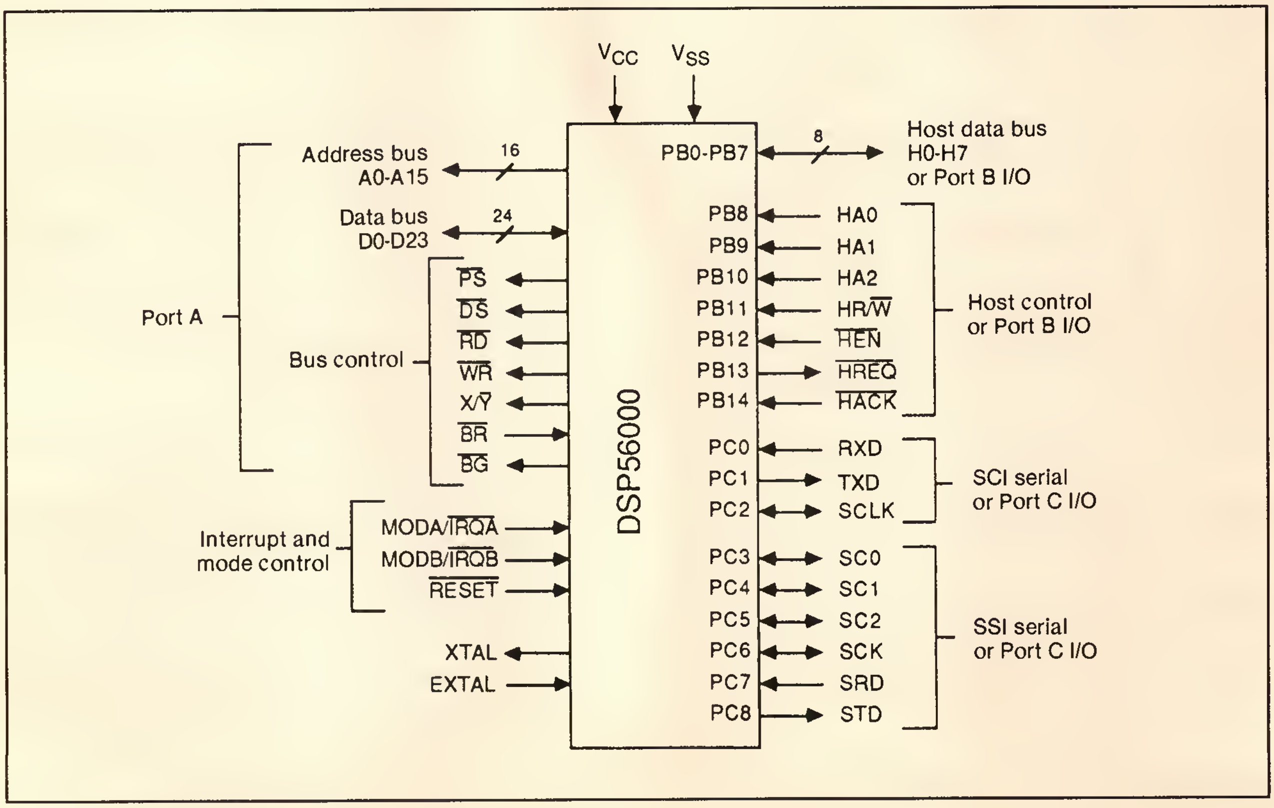


Figure 9. DSP56000 pinout.

peripherals and slave processors (microprocessors or another DSP IC) may also be accessed through this port. Separate 16-bit address and 24-bit data buses are used to multiplex the three internal address buses and four internal data buses off-chip, respectively. Off-chip (external) memory spaces are a logical extension of on-chip (internal) memory spaces. The bus controller determines, from the value of the address, whether a memory access is external or internal and schedules the bus activity to optimally use the memory expansion bus. If only one external memory access is requested per instruction cycle, the request is granted immediately and no extra clock cycles are required. If two or three external memory accesses are requested in a given instruction cycle, a minimum of one or two extra instruction cycles, respectively, are required to complete the instruction. The seven bus control signals create a synchronous bus that can perform 10.25 million accesses per second. Full-speed operation with no wait states requires a memory access time of 55 ns. The bus controller can be programmed to insert 0 to 15 wait states for four types of external memory access. Each wait state is one clock cycle (or 48.75 ns) long with a 20.5-MHz processor clock. This allows fast and slow external devices to be mixed on the memory expansion bus. An external device can also gain control of the memory expansion bus by asserting the bus request (BR) control input. In response to a BR, the DSP56000 releases control of the bus and asserts the bus grant (BG) control output at the end of the current bus access.

General-purpose I/O pins. On the DSP56000, 24 pins can be programmed to be general-purpose I/O pins; these pins are called Port B and Port C (see Figure 9 again). When they are programmed in this way, they can be used as I/O flags for synchronization and control purposes. Individual pin control, data direction, and data transfer functions are provided by six internal memory-mapped registers. The user can change or test these registers using standard bit manipulation and jump-on-bit-condition instructions. Each port pin can also be programmed to serve as a dedicated pin for one of the three on-chip peripherals.

Host processor interface. Although the DSP56000 can operate as a stand-alone processor, in many systems it is accompanied by a host microprocessor. The host processor functions as the system controller and user interface, while the DSP56000 handles the real-time digital signal processing. To support its use in such multiple-processor systems, the DSP56000 includes an on-chip host processor interface. The host interface is a byte-wide, full-duplex parallel port that can be connected directly to the data bus of a host processor. The host processor may be any of a number of industry-standard microcomputers or microprocessors, another DSP IC, or DMA hardware. The host interface appears as a memory-mapped peripheral occupying eight bytes in the host processor address space. It gives the host processor an eight-bit bidirectional data bus and seven control lines to use to control data transfers. It provides 14 internal registers to support double-buffered data transfers between the host processor and the core processor by means of asynchronous polling or interrupts. In DMA mode, the host interface allows an external DMA controller to perform DMA transfers between an external memory or device and the host interface's registers using the host request (HREQ) and host acknowledge (HACK) handshake lines. From the DSP56000's perspective, the DMA data can be transferred between the host interface registers and any DSP56000 register or memory location (internal or external) by means of the standard instruction set. The fast interrupt mechanism can be used to minimize the data transfer overhead.

A major accomplishment of the DSP56000 design was low power consumption in both active and standby modes.

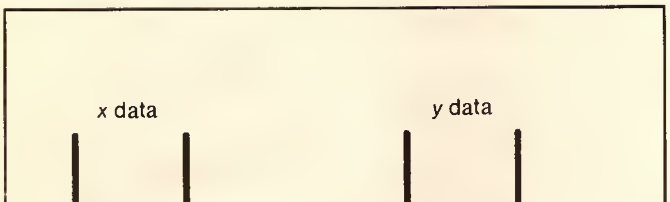
The host interface provides DMA initialization commands that are used to set up the host interface DMA channel. It also has a special host command feature that allows the host processor to issue a vectored interrupt request to a DSP56000 program. The host processor may select one of 32 DSP56000 host command interrupts by writing a vector address register. Host commands are useful for debugging, performing on-line diagnostics, implementing control protocols, and setting up DMA.

Serial communications interface. The serial communications interface, or SCI, provides full-duplex serial communications with a variety of serial devices—including microprocessors, other DSP ICs, terminals, and modems—either directly or via RS-232 lines. The SCI supports industry-standard asynchronous character modes and allows parity and multidrop options. The multidrop option includes wake-up-on-idle-line and wake-up-on-address-bit capabilities. A synchronous shift register mode allows I/O expansion and high-speed, synchronous data transmission. The SCI consists of separate transmit and receive sections and a programmable baud-rate generator. Seven internal registers provide doubled-buffered data transfer and control functions. Three I/O pins are used for transmit data, receive data, and baud-rate clock functions. The baud rate can be internally or externally generated for asynchronous rates of up to 320K bits per second and synchronous rates of up to 2.5M bits per second. The internal baud-rate generator can function as a periodic interrupt timer when it is not being used by the transmit and receive sections.

Synchronous serial interface. The synchronous serial interface, or SSI, provides a full-duplex, double-buffered

Table 7.
DSP56000 benchmark summary.

Benchmark	Performance
N-tap real FIR filter with data shift	97.5 ns per tap
N-tap real, LMS adaptive FIR filter with data shift	292.5 ns per tap
N real, cascaded IIR biquad filters (four coefficients)	390 ns per filter
N-tap complex FIR filter with data shift	390 ns per tap
256-point complex FFT (radix 2, looped)	0.706 ms
1024-point complex FFT (radix 2, looped)	4.994 ms
Two-dimensional convolution (3 × 3 coefficient mask)	975 ns per output
Finding of maximum absolute value and index in array	195 ns per point



```
start   move    #data,r0
        move    #coef,r4
        move    #N-1,m0
        move    m0,m4

fir     movep                   x:input,x:(r0)
        clr     a               x:(r0)-,x0      y:(r4)+,y0
        rep     m0
        mac     x0,y0,a         x:(r0)-,x0      y:(r4)+,y0
        macr    x0,y0,a         (r0)+
        movep                   a,x:output
        jmp     fir
```

Figure 10. DSP56000 finite impulse response (FIR) filter—data memory organization (top) and program (bottom).

serial port that allows the DSP56000 to communicate with a variety of serial devices. These include one or more industry-standard codec A/D and D/A converters, other DSP ICs, microprocessors, and serial peripheral devices. The SSI consists of separate transmit and receive sections and an SSI clock generator. The clock generator defines the serial bit rate, the serial word size, and the number of serial words per frame. The data in each serial frame are controlled by software, allowing any user protocol to be implemented. Several clock and frame sync timing options provide flexible, synchronous, serial communications at rates of up to 5M bits per second. The SSI uses three to six I/O pins, depending on the operating mode; it has eight internal registers.

Three SSI operating modes support the requirements of different serial devices. The normal operating mode is used for periodic devices that transmit or receive one data word with each serial frame. One time slot is defined for data transmission at the start of each serial frame. A codec A/D or D/A converter is an example of such a periodic device. The on-demand operating mode is for nonperiodic communications such as those from one DSP56000 to another. No time slots are defined for transmission and data are transmitted as soon as they are available. The network operating mode defines from 2 to 32 time slots per serial frame which can be used for creating a network of communicating serial devices. Each device can transmit or receive during one or more assigned time slots. The time slot assignments for each serial device are determined by the user's software. In network mode, multiple DSP56000s can communicate without needing glue chips to do so. In network mode, the DSP56000 can also be interfaced directly to the time-division-multiplexed, serial I/O channels used in telecommunications applications.

DSP56000 implementation

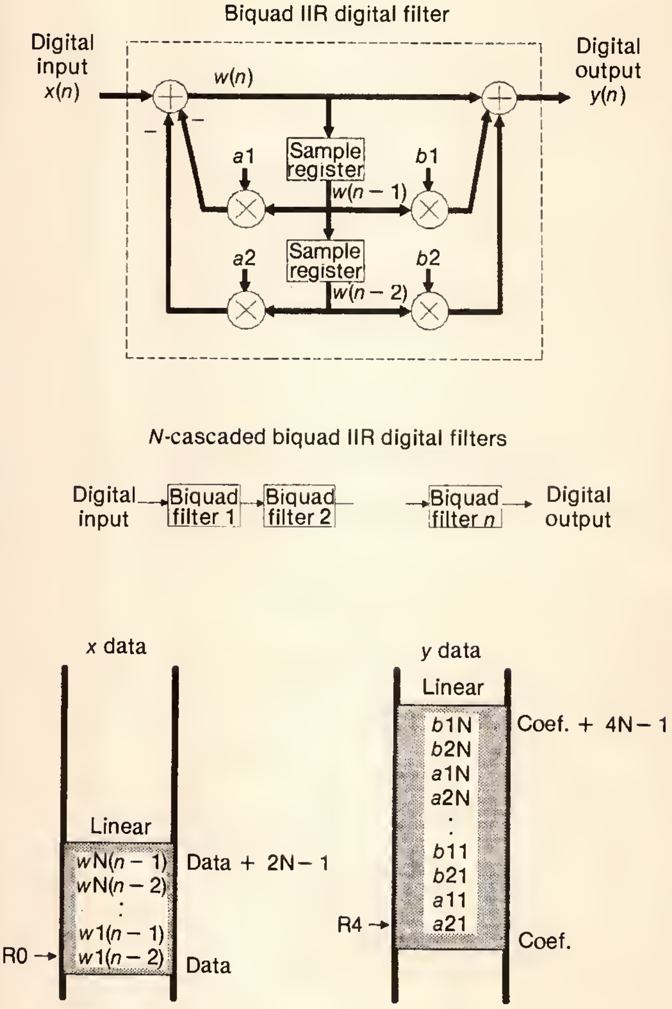
The DSP56000 is implemented in a 1.5-μm, double-level metal, N-well, high-density CMOS process. It is an 88-pin integrated circuit available in pin-grid-array or surface-mount packaging. Its maximum clock rate is currently 20.5 MHz and is provided by an on-chip crystal oscillator or an external clock. One of the major design accomplishments was low power consumption in both active and standby modes. Because of the DSP56000's full CMOS design, its power consumption scales down linearly with clock frequency, allowing the user to reduce processor speed to save power. Since the DSP56000 contains substantial on-chip memory, the user can locate program and data sections with high dynamic frequency in on-chip memory to avoid driving the memory expansion port. The DSP56000's bus controller does not toggle the external address and data bus pins unless an external access is made. The DSP56000 deselects all external devices to their low-power standby modes when they are not being accessed. It also saves power by using STOP and WAIT low-power standby modes. Execution of a STOP or WAIT instruction puts the DSP56000 into a low-power standby mode while the DSP system is off-line or waiting for an interrupt to occur, respectively. This "software power control" technique is popular in the MC146805 and MC68HC11 microcomputer families.

Software examples and performance analysis

The DSP56000's performance of some common DSP benchmarks is summarized in Table 7. However, the performance and software features of the DSP56000 can best be demonstrated by several typical DSP software examples.

Finite impulse response filter. The sample FIR digital filtering application of Figure 1 is called an *N*th-order real filter, where *N* is the number of coefficients and "real" means that the data and coefficients are real numbers. The DSP56000 assembler source program and memory organization for this system are shown in Figure 10. The *N* data samples are stored in *x* data memory and the *N* filter coefficients are stored in *y* data memory. This natural partitioning is desirable since both *x* memory and *y* memory space can be accessed in the same instruction cycle. Both the data and coefficients are stored in modulo *N* buffers, but for different reasons. Modulo *N* data addressing is used to time-shift the simulated shift register by an incrementing of R0 by 1 after the FIR calculation. Modulo *N* coefficient addressing is used for convenience for automatically wrapping around, at each sampling period, the address pointer R4 to the first coefficient. The four instructions at START are used to set up the two modulo *N* address pointers R0 and R4. The seven instructions at FIR form a simple loop to get an "input" sample, perform the FIR filtering operation, and store the filter "output." The first data and coefficient are preloaded into the data ALU while the accumulator is cleared (CLR). The actual filtering operation is performed by repeating (REP) the multiply/accumulate (MAC) instruction. The last tap of the filter is performed by a multiply/accumulate and round (MACR) instruction to form a single-precision, rounded result. All calculations are performed to 56-bit precision and no intermediate data are lost. The accumulator extension register protects against overflows for large *N*. The FIR filtering kernel is performed in one instruction cycle per tap. A complete *N*th-order, real FIR filter executes in $N+3$ instruction cycles. For a 32nd-order filter, a 20.5-MHz DSP56000 can process a sample frequency (*fs*) as high as 250 kHz in real time.

Infinite impulse response filter. An IIR filter employs feedback paths to achieve an infinite impulse response and generally provides the most filtering capability for a given computational load and storage. One of the most popular IIR filters is the second-order section, or real biquad, filter. The biquad filter has two poles and two zeroes in its transfer function and is the digital equivalent of a two-pole analog filter. The biquad filter is often connected in series (cascaded) to form higher-order digital filters. Figure 11 shows the biquad filter's block diagram, its data memory



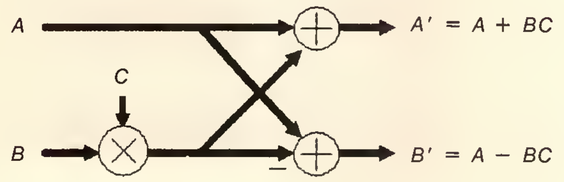
```
start   move    #$ffff,m0
        move    m0,m4

iir     move    #data,r0
        move    #coef,r4
        movep                   x:input,a
        move                    x:(r0)+,x0          y:(r4)+,y0

        do      #N,end_iir
        mac     -x0,y0,a        x:(r0)-,x1          y:(r4)+,y0
        macr    -x1,y0,a        x1,x:(r0)+          y:(r4)+,y0
        mac     x0,y0,a         a,x:(r0)+           y:(r4)+,y0
        mac     x1,y0,a         x:(r0)+,x0          y:(r4)+,y0
end_iir
        rnd     a
        movep                   a,x:output
        jmp     iir
```

Figure 11. DSP56000 infinite impulse response (IIR) filter—biquad IIR filter diagrams (top), data memory organization (middle), and program (bottom).

organization, and the DSP56000 assembler source program needed to implement N cascaded biquad digital filters. The memory is organized to store the filter storage $w(n-1)$ and $w(n-2)$ in x data memory and the filter coefficients $a1$, $a2$, $b1$, and $b2$ in y data memory. In this example, the time-shift function is performed by parallel move operations instead of by modulo addressing. The two instructions at START set up data pointer R0 and coefficient pointer R4 for linear arithmetic. The program loop at IIR loads a digital "input," performs N cascaded biquad filters, and stores the filter "output." The actual biquad filter program consists of only four multiply/accumulate instructions located within a hardware DO loop. The DO instruction initiates the hardware DO loop with the instruction following the DO instruction and ends it with the instruction before the label END_IIR. The digital output is rounded (RND) to single precision after the hardware DO loop has executed. The four-coefficient, biquad filter kernel executes in only four instructions, i.e., in 390 ns per biquad filter. Similar five-coefficient biquad filter kernels execute in five instructions, or 487.5 ns. For typical voiceband processing with an 8-kHz sampling rate, the DSP56000 can implement over 300 biquad digital filters in real time.

Fast Fourier transform. The fast Fourier transform, or FFT, is widely used in signal processing to perform spectral analysis of time-domain data. Most FFTs operate on complex data having real and imaginary components and hence must use complex arithmetic. The DSP56000 is very efficient at complex arithmetic. Complex arithmetic generally requires at least four multiplier input registers to store two complex inputs, two accumulators to store the real and imaginary parts of the complex result, and address pointers that can access complex data pairs. A large number of FFT algorithms exist, but here we will demonstrate only the simple, radix-2, decimation-in-time, in-place complex FFT. The heart of the radix-2 FFT is a complex "butterfly" kernel, which is shown in Figure 12. Note that all data paths and calculations are done in complex arithmetic. This butterfly kernel is executed many times to calculate a complete FFT. The complete DSP56000 macro program for a radix-2, decimation-in-time, in-place complex FFT is also shown in Figure 12. This macro may be called to perform an FFT of any size from 2 to 32,768 points. The example uses three nested hardware DO loops to compact the complete FFT program into only 40 words of program memory. The butterfly kernel routine inside the innermost DO loop requires only six instructions, or 585 ns, to perform one butterfly calculation on two complex data points A and B. The single complex multiplication (B × C) is implemented as four real multiplies. The inner, middle, and outer DO loops build on the kernel routine to process one butterfly group, one butterfly pass, and the complete FFT transform, respectively. This radix-2 FFT using looped code executes a 256-point complex FFT in 814 μs and a 1024-point complex FFT in 6.41 ms. Other variations achieve even higher performance. For instance, modified versions of the code in the example execute a 1024-point complex FFT in less than 5 ms.

Radix-2, decimation-in-time, complex FFT butterfly



Note: All variables and arithmetic are complex.

```
fft     macro   points,data,coef
        move    #points/2,n0
        move    #1,n2
        move    #points/4,n6
        move    #-1,m0
        move    m0,m1
        move    m0,m4
        move    m0,m5
        move    #0,m6
;
; Perform all FFT passes with triple nested DO loop
;
        do      #@cvi(@log(points)/@log(2)+0.5),_end_pass
        move    #data,r0
        move    r0,r4
        lua     (r0)+n0,r1
        move    #coef,r6
        lua     (r1)-,r5
        move    n0,n1
        move    n0,n4
        move    n0,n5

        do      n2,_end_grp
        move                        x:(r1),x1       y:(r6),y0
        move                        x:(r5),a        y:(r0),b
        move                        x:(r6)+n6,x0
;
; Radix 2, decimation in time, complex FFT butterfly kernel
;
        do      n0,_end_bfy
        mac     x1,y0,b                             y:(r1)+,y1
        macr    -x0,y1,b            a,x:(r5)+       y:(r0),a
        subl    b,a                 x:(r0),b        b,y:(r4)
        mac     -x1,x0,b            x:(r0)+,a       a,y:(r5)
        macr    -y1,y0,b            x:(r1),x1
        subl    b,a                 b,x:(r4)+       y:(r0),b
_end_bfy
        move                        a,x:(r5)+n5     y:(r1)+n1,y1
        move                        x:(r0)+n0,x1    y:(r4)+n4,y1
_end_grp
        move    n0,b1
        lsr     b       n2,a1
        lsl     a       b1,n0
        move    a1,n2
_end_pass
        endm
```

Figure 12. DSP56000 fast Fourier transform (FFT)—diagram of the "butterfly" kernel (top) and program (bottom).

Since the FFT data are stored in 24-bit words, little or no scaling is required with typical 12-to-16-bit input data. The DSP56000 scaling modes can be used to scale the data of each FFT pass one bit left or right with minimum overhead. This can add 20 more bits of dynamic range to a 1024-point FFT while still maintaining full FFT speed.

DSP56000 design tools

The DSP56000 is supported by a software development package consisting of a full-featured macro cross assembler, ASM56000, a software simulator, SIM56000, and associated documentation. The package is available for the IBM PC/MS-DOS, VAX/VMS, and Unix environments. ASM56000 offers the usual complement of features found in modern assemblers, such as extensive error reporting, conditional assembly, file inclusion, nested macros with macro library support, local labels, sections, and external definition/reference directives. It also provides arbitrary expression evaluation with Boolean operators and built-in functions for data type conversion, string comparison, and common transcendental functions such as sine, cosine, logarithm, exponent, and square root. These functions allow constants and lookup tables for DSP algorithms to be parameterized by macro arguments and dynamically generated at assembly time. ASM56000 also provides assembler output listings with instruction cycle counts, cross reference tables, and memory utilization reports. The memory utilization reports provide a global view of the allocated and free space in the DSP56000 memory map, which can become fragmented by inefficient placement of modulo and reverse-carry (bit-reversed) storage regions. A report identifies the free space available for additional storage in each memory map.

Digital signal processors are transforming analog circuits into software in the 1980's in the same way microprocessors transformed digital control logic into software in the 1970's.

The SIM56000 software simulator emulates, on a clock cycle basis, the functions of the DSP56000, including on-chip peripheral activity and external I/O pin activity. SIM56000 enables the software developer to execute DSP56000 object code generated from ASM56000 or from SIM56000's own single-line assembler. Throughout the DSP56000 chip design, SIM56000 was used to compare against the chip data base simulations by running the same DSP56000 programs through both simulations. Both the chip design and the software products were debugged rapidly through this comparison testing. SIM56000 performs, on an instruction or clock cycle basis, single stepping or tracing with multilevel conditional or unconditional breakpoints. It provides instruction and clock cycle counts and generates histograms of those counts to support the analysis of program execution time. It displays the enabled set of registers and memory, highlights the write operations, and provides options for displaying upon a read, write, or other access. Of particular note are SIM56000's input and output commands, which assign the simulator's I/O, with a device pin, memory location, or on-chip peripheral, to a terminal or a disk file. Its I/O data format can be untimed or "time stamped," and nested repeat directives can be used to generate arbitrary input data sequences. Besides being an accurate simulator of the DSP56000, SIM56000 is very easy to use. Help files describe each simulator command and a help line on the display indicates the command line syntax as commands are entered. A symbolic calculator assists the programmer with hexadecimal, decimal, and binary calculations. The programmer may define, store, and execute simulator command macros. He can also do program patching by using SIM56000's single-line assembler/disassembler. He can log all simulator activity to disk files for future analysis, and he can save the simulator state so he can resume the simulation later. SIM56000 uses ASCII disk files for all of its I/O (but not for saving the simulator state), enhancing access by other programs and utilities.

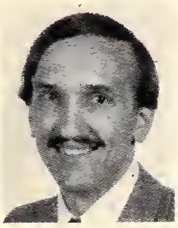
Motorola is developing a DSP56000 evaluation module, or EVM, to serve as a low-cost design tool. The EVM consists of an evaluation board, or EVB, an interface card for the IBM PC bus, and a user interface software package, EVM56000. The EVB contains a full-speed 20.5-MHz DSP56000, 8192 × 24 bits of external program/data RAM, an expansion connector for adding prototype hardware, and a monitor ROM. It is controlled by the EVM56000 software running in the PC environment, which presents a user interface similar to that of the SIM56000 software simulator. EVM56000 retains many of SIM56000's features and adds commands to support up to eight EVBs on the same host with foreground/background access to the user interface. The EVB can be used as a hardware accelerator for speeding up simulations or as a prototype board for developing target applications.

Additional software support includes software source libraries, application notes, and a high-level-language compiler. Moreover, Motorola supports the user with training seminars, video classes, and an electronic bulletin board.

The DSP56000 is a state-of-the-art, high-performance digital signal processor. The similarities between it and other Motorola microprocessors make it easy to learn and easy to program, yet the differences open up new user applications. The future for programmable digital signal processors looks extremely bright. Digital signal processors are transforming analog circuits into software in the 1980's in the same way microprocessors transformed digital control logic into software in the 1970's. By the 1990's, most communications, computer, and control equipment will use digital signal processing ICs. DSP ICs will evolve as microprocessors have. Work continues on developing an entire family of faster and more capable DSP devices to serve the needs of DSP users.

References

1. "Motorola's Sizzling New Signal Processor," *Electronics*, March 10, 1986, pp. 30-33.
2. J. Bates, "Motorola's DSP56000: A Fourth Generation Digital Signal Processor," *Professional Program Session Record, 1986 Mini/Micro Northeast Computer Conf.*, No. 18, Paper 5.
3. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1975.
4. R.C. Jaeger, "Tutorial: Analog Data Acquisition Technology, Parts 1-4," *IEEE Micro*, May 1982, pp. 20-37, Aug. 1982, pp. 46-56, Nov. 1982, pp. 20-35, and Feb. 1983, pp. 52-61.
5. L.R. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
6. S. Abiko et al., "Architecture and Applications of a 100-ns CMOS VLSI Digital Signal Processor," *Proc. IEEE ICASSP 86 Conf.*, pp. 393-396.
7. T. Nishitani et al., "Advanced Single-Chip Signal Processor," *Proc. IEEE ICASSP 86 Conf.*, pp. 409-412.
8. J.R. Boddie et al., "The Architecture, Instruction Set and Development Support for the WE DSP32 Digital Signal Processor," *Proc. IEEE ICASSP 86 Conf.*, pp. 421-424.
9. K.A. El-Ayat and R.K. Agarwal, "The Intel 80386—Architecture and Implementation," *IEEE Micro*, Dec. 1985, pp. 4-22.
10. D. Phillips, "The Z80000 Microprocessor," *IEEE Micro*, Dec. 1985, pp. 23-36.
11. K.L. Kloker, "Architectural Features of the Motorola DSP56000 Digital Signal Processor," *VLSI Signal Processing, II* (Proc. 1986 IEEE Workshop on VLSI Signal Processing), IEEE Press, New York, 1986.
12. *DSP56000 Digital Signal Processor User's Manual*, Pub. no. DSP56000UM/AD, Motorola, Inc., Austin, Texas, 1986.



Kevin L. Kloker, a principal staff engineer in Motorola's Corporate Research Laboratories, was the principal architect of the Motorola DSP56000 programmable digital signal processor. His areas of interest include VLSI processor architectures, computer arithmetic, software languages, and digital signal processing. He received a BSEE from Bradley University and a MSEE from the Illinois Institute of Technology in 1976 and 1980, respectively, and joined Motorola in 1976. He is a member of the Motorola Science Advisory Board Associates, the IEEE, Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

Questions about this article can be directed to Kloker at Motorola, Inc., 1301 East Algonquin Road, Schaumburg, IL 60196.

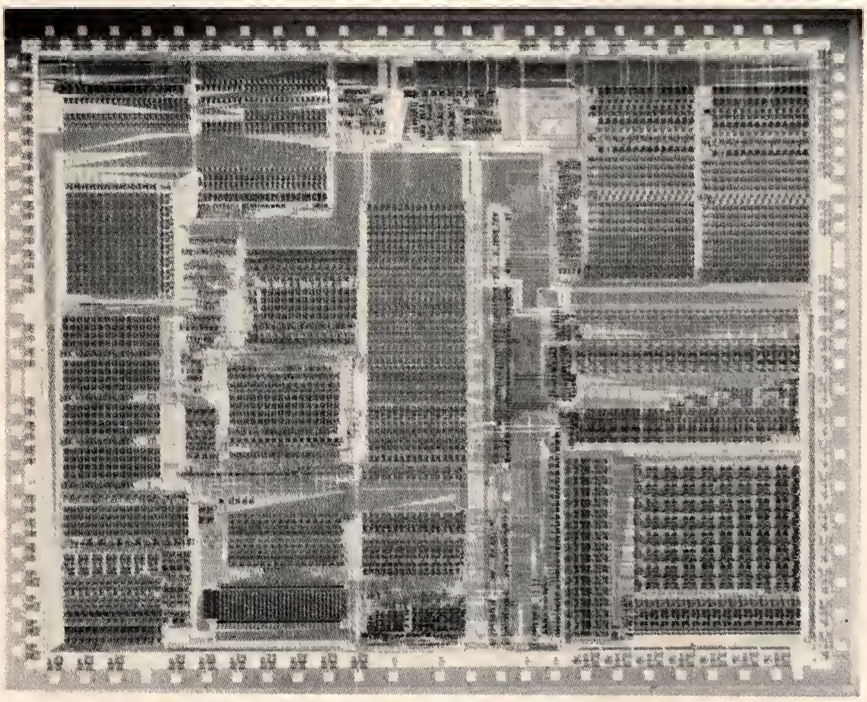
Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 177 **Medium** 178 **Low** 179

The ADSP-2100 DSP Microprocessor

John P. Roesgen
Concord Data Systems



The 2100 accesses external memory efficiently and devotes its silicon area to providing greater functionality and processing throughput.

The performance of single-chip digital signal processors has always lagged far behind the requirements of many application areas. There are several reasons for this discrepancy, some related to the limits of VLSI techology and others to architectural considerations. However, given the inherent ease of use and cost advantages that these devices offer, there is a strong desire to extend their application into high-performance areas.[1]

One key factor that affects DSP performance is the interface between the processor and its memory. To date, most DSP chips dedicate a large portion of their silicon area to on-chip memory. A design of this sort not only constrains the size of the memory it also confines the processing logic to a smaller area and reduces its functionality. Furthermore, it usually limits access to external memory and in some cases incurs a speed penalty.

The Analog Devices ADSP-2100 microprocessor represents an alternative approach to digital signal processing architecture. Unlike several other single-chip DSPs, the 2100 is designed to access external memory efficiently. With the exception of a small instruction cache, the chip itself contains no memory. The enormous amount of silicon real estate saved by excluding the memory is used to add functionality and increase processing throughput significantly beyond that of previous single-chip designs.[2]

The 2100 includes such items as a full-function barrel shifter for normalization and denormalization, two independent data address generators with modulo addressing capability, a program sequencer with provisions for zero-overhead looping, a background register set for rapid context switching, and sufficient internal busing to support a high degree of parallelism in the instruction set. An advanced 1.5-micrometer CMOS process gives the chip an instruction cycle time of 125 nsec and a power consumption of less than 1/2 watt.

0272-1732/86/1200-0049$01.00 © 1986 IEEE

System configuration

Figure 1 shows a basic ADSP-2100 system configuration. The processor interfaces with two external memory systems, a program memory and a data memory. As the names suggest, program memory holds the application program and data memory holds the system data. Because they are separate, instructions and data can be accessed simultaneously. Data can also be stored in the program memory, which also allows dual data access.

A set of address, data, and control lines is provided for each memory. On the program memory side there are 14 address lines (PMA), 24 data lines (PMD), a memory select signal (PMS), read and write strobes (PMRD and PMWR), and a signal to indicate when data (as opposed to an instruction) is being accessed (PMDA). The 14 address lines give an address range of 16K words, which can be expanded to 32K if the PMDA signal is used as an additional address bit.

On the data memory side there are 14 address lines (DMA), 16 data lines (DMD), a memory select (DMS), read and write strobes (DMRD and DMWR), and a signal to acknowledge the transfer of data (DMACK). Peripheral devices are memory mapped into the data memory address space. Slower devices can stretch the memory cycle as needed by withholding the DMACK signal.

The chip supports multiprocessing applications with bus-request and bus-grant signals (BR and BG). The 2100 responds to a bus request by halting program execution and releasing the address, data, and control lines to the memories so that another processor can access them directly.

Four interrupt request (IRQ) inputs are provided for external devices that need periodic service from the processor. The interrupt pins can be individually programmed for either level or edge sensitivity. The four inputs are prioritized with options for nesting (higher priority levels interrupting lower ones) or blocking (only one level serviced at a time). The maximum response time for an unmasked interrupt request is two cycles.

A high-level internal block diagram of the 2100 is shown in Figure 2. Three separate computational units are provided, an ALU, a multiplier-accumulator (MAC), and a barrel shifter. Together, these offer a wide variety of fast arithmetic functions. Two independent data address units generate the external memory addresses that keep data flowing between computation and memory. The program sequencer coupled with an on-chip cache memory maintains a continuous instruction stream to the rest of the processor. Five major buses speed the transfer of information between the various functional blocks. These buses provide the necessary paths to allow the execution of complex multifunction instructions in one machine cycle. Four of them extend off chip to become the address and data lines for the external memories.

Computational features

The computational section of the processor is divided into three independent units. Rather than being arranged in the usual series fashion, these units rest side by side, relying on the R bus as a flexible interconnect path. Operation of the R bus allows any sequence of arithmetic operations to be performed smoothly, without excessive juggling of intermediate results.

The 16-bit-wide ALU performs general-purpose arithmetic and logical operations. The arithmetic functions include add, subtract, negate, increment, decrement, absolute value, and divide. Provisions are included for both double-precision and saturation arithmetic. The available logic functions are AND, OR, Exclusive OR, and NOT.

The MAC performs multiply, multiply-accumulate, and multiply-subtract operations. The 16×16-bit multiplier array produces a 32-bit product, which is fed into a 40-bit adder/subtracter. The final, 40-bit-wide result leaves plenty of room for overflow. Multiplier inputs can be any combination of signed or unsigned formats, making double-precision multiplication possible. Options also exist for unbiased rounding and saturation of the final result.

The shifter efficiently implements the numerical scaling operations needed for floating-point arithmetic. These operations include normalization, denormalization, shifting by a constant, and deriving an exponent for an individual number or block of numbers. The shifter array accepts a 16-bit input and produces a 32-bit output. Both zero-filling and sign extension of the result are available. Multiprecision shifting operations are also fully supported.

Each of these arithmetic units contains a set of input and output registers, which act as stopover points for data as it moves between the external memory and the computational circuitry. The registers therefore introduce a level of pipelining into the dataflow. The processor's instruction set accommodates this capability by allowing computations and register-memory transfers to be overlapped. Computational operations take their operands either from the local input registers or from an output register via the R bus and then load the result into a local output register.

Register names are derived from their function to ease the programming task. For example, AX0 and AX1 are the ALU X input registers, AY0 and AY1 are the ALU Y input registers, and AR is the ALU result register. The equivalent registers in the MAC are MX0, MX1, MY0, MY1, and MR. The shifter's register names are SI for the input register, SR for the result register, and SE for the exponent register.

A complete set of background input and output registers can be activated at any time if the processor must change tasks quickly. This capability effectively doubles the number of available registers and can eliminate the save and restore overhead associated with context switching. So for example, execution of interrupt service routines that require the computational facilities of the processor can be sped up tremendously. By switching to the background register set, the processor can save its current computational state in one cycle. Switching back to the original registers will then restore the previous context at a later time.

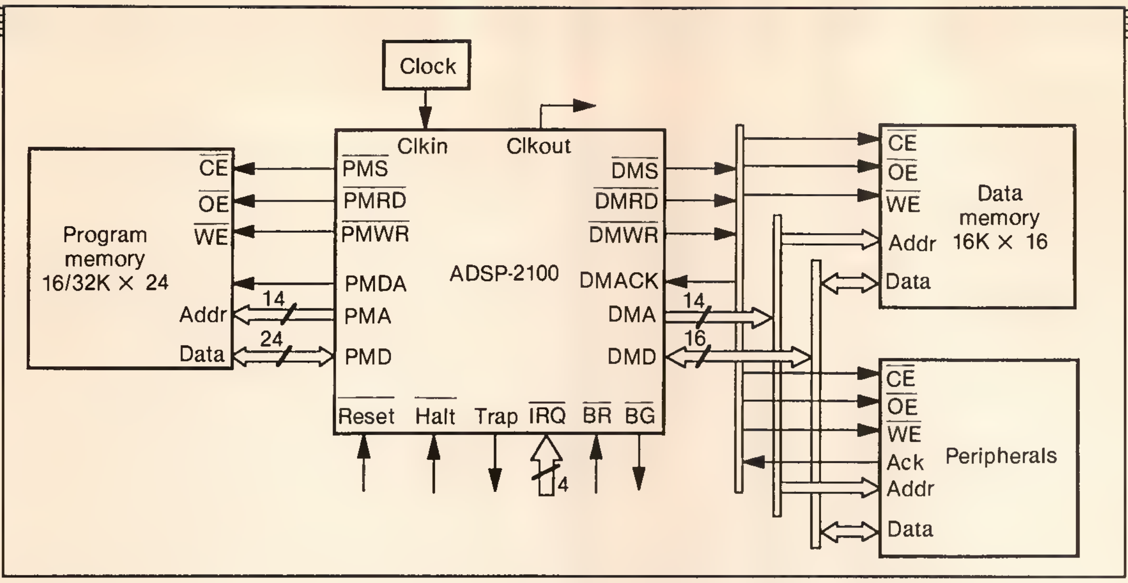


Figure 1. ADSP-2100 system diagram.

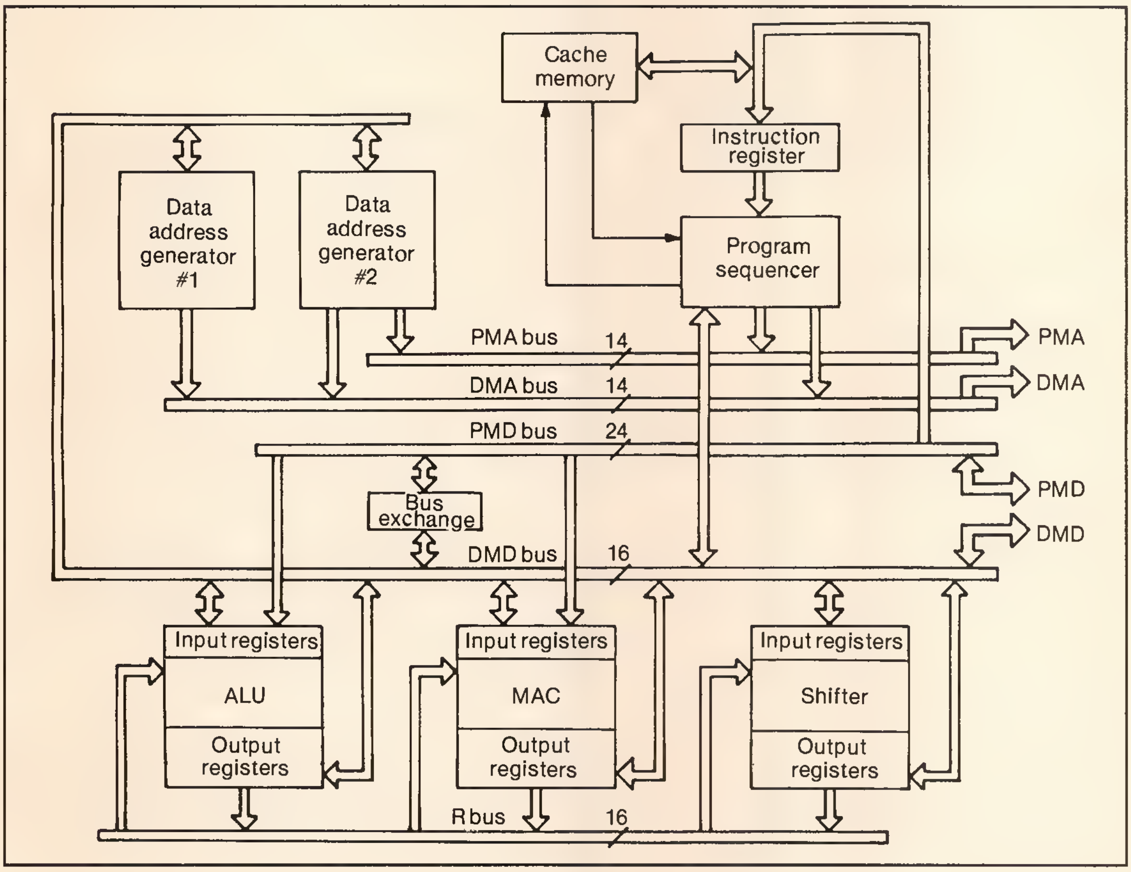


Figure 2. ADSP-2100 internal block diagram.

Address generation

Fast number-crunching hardware is of little benefit if it must frequently sit idle waiting for data. A powerful memory-addressing scheme prevents this situation and keeps memory references going at a rate equal to the processing rate. Should an operation (such as an add or multiply) require two operands, both must be supplied at this rate.

The 2100 contains two independent address generators. Both supply data memory addresses, but one of them can also address the program memory, allowing access to "data" stored there as well. Thus the processor has the capability of fetching two operands simultaneously, one from data memory and one from program memory. One obvious application is digital filtering. By storing samples in one memory and coefficients in the other, the processor can access both in a single machine cycle and keep the multiplier-accumulator running at full speed.

Each address generator contains the elements shown in Figure 3. Memory pointers are kept in the I (index) register file. The M (modify) register file contains incremental values, which move the pointers by a desired amount each time they are used. The L (length) registers define the size of each data structure being accessed. Each of the register files contains four 14-bit registers, which are loadable and readable via the internal DMD bus. Address generator 1 also has a bit-reversing capability to aid in the scrambling or unscrambling of data in fast Fourier transforms.

Whenever an indexed memory reference is made, a selected I register provides the address. An independently selected M register is then added to the address to form a tentative next address. The tentative next address feeds into the modulus logic along with the selected L register value. The modulus logic determines whether the new address is outside the bounds of its associated data structure. If it is, the address wraps around in a modulo fashion to remain within its allowable range. Otherwise, the address passes through the modulus logic unchanged. In either case the output of the modulus logic is loaded back into the original I register, ready for the next memory reference. The complete address modification process can be described with the following formula:

$$\begin{aligned}\text{Next address} &= (\mathrm{I}+\mathrm{M}-\mathrm{B})\text{modulo}(\mathrm{L}) + \mathrm{B}\\ \mathrm{I} &= \text{Index register value}\\ \mathrm{M} &= \text{Modify register value}\\ \mathrm{B} &= \text{Base address}\\ \mathrm{L} &= \text{Length register value}\end{aligned}$$

Notice that this computation requires the memory base address but that it is not supplied. This information is implied indirectly by adopting the following two rules:

- If the buffer length requires *n* bits to be represented in binary, the lower *n* bits of the buffer base address must be zero, and
- The modify value M should not be greater than the length L.

With these restrictions, the buffer base address B can be extracted from the I + M value by masking out the lower *n* bits and setting them to zero.

As an example of modulo addressing, suppose that an I register points to the last location in a circular buffer of length 5. Then modifying the pointer with an M register containing +1 actually decrements the address by 4, back to the first location in the buffer. If the pointer is then modified by an M register containing −2, the address increments by 3 and points to the next-to-last location. Of course if a pointer modification does not cross either boundary, it behaves in the usual fashion.

Cache memory

Fetching data from program memory would seem to be in conflict with the normal instruction fetches that keep the program going. One way for the processor to deal with this conflict is for it to insert an extra memory cycle for the data fetch. But this method negates the advantage of storing the data in program memory, since it could have just as easily been fetched from data memory during an extra cycle.

Fortunately, most time-critical computations are repetitive in nature. The 2100 executes these computations in the form of program loops, and this is where the on-chip cache memory comes in. The job of the cache memory is to maintain a small (16-word) history of previously executed instructions. When the program enters a loop, the cache stores the loop instructions on the first pass, but on all subsequent passes it can feed the instruction register and free the program memory for data fetches without incurring extra cycles. Thus the processor's performance under these conditions approaches that of a three-memory system. The processor maintains the cache memory and makes the extra cycle decision during execution, making them transparent to the user.

Program sequencer

Keeping numerical throughput high also requires a sophisticated program sequencer, for if the processor gets bogged down by branching, looping, or responding to interrupts, the computation rate suffers. A large portion of the 2100 chip area was dedicated to the program sequencer to streamline the program flow and minimize overhead. Figure 4 shows a detailed block diagram of the program sequencer.

Instruction addresses can come from four possible sources: a 14-bit program counter (PC), an internal 16-level PC stack, an interrupt controller, or a 14-bit field of the instruction register. The program counter keeps track of the current instruction address and feeds an incrementer, which provides the next contiguous address. The PC stack stores subroutine and interrupt-return addresses and is chosen when returning to main program execution. The interrupt controller monitors the external interrupt-request inputs and provides jump vectors when servicing is needed. The instruction register is chosen when a direct jump is executed.

The 2100 includes status registers to keep track of arithmetic results, execution modes, and interrupt configuration. Arithmetic status drives the condition logic that controls the selection of a next address for conditional operations. Interrupt configuration status transfers to the interrupt controller. An internal four-level-deep stack saves status information automatically when vectoring to an interrupt service routine and restores it upon return. The status stack can also be pushed or popped manually at any time.

The down counter controls program looping with a decrement and branch feature. Preloaded via the internal DMD bus, it generates a counter-expired (CE) status output when the count reaches zero. Decrementing occurs automatically every time the status is checked. A four-level stack associated with the counter allows counted loops to be nested five levels deep.

The loop stack and comparator also facilitate program looping. The do-until instruction sets up these functions.

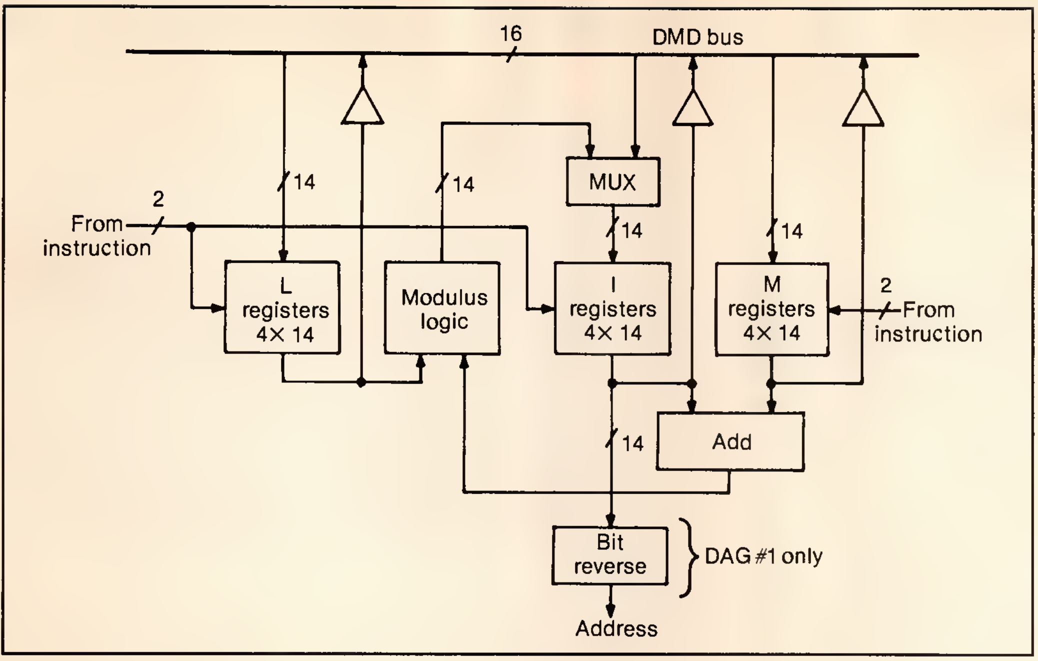


Figure 3.
Address generator block diagram.

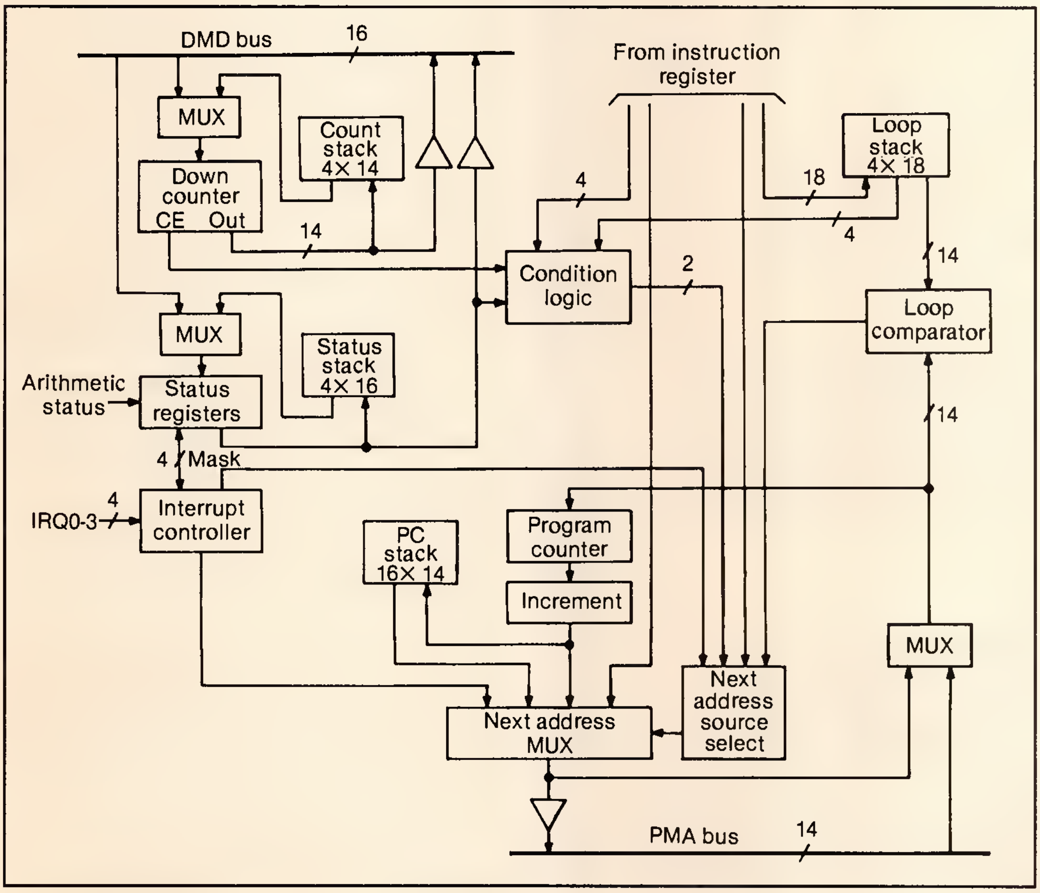


Figure 4.
Program sequencer block diagram.

When executed, this instruction pushes the end-of-loop address and termination condition onto the loop stack and the beginning-of-loop address (PC + 1) onto the PC stack. Once the loop is entered, the loop comparator compares the next address output of the sequencer with the end-of-loop address on the loop stack. When the two are equal, it indicates that the processor is fetching the last instruction in the loop. During the next cycle, while the last instruction is being executed, the condition logic tests the termination condition specified by the loop stack. If the termination condition is false, the sequencer jumps back to the beginning of the loop by choosing the PC stack as the next address. Otherwise, the sequencer exits the loop by choosing PC + 1 as the next address.

Move instructions
- Register ↔ register
- Register ↔ data memory
- Register ↔ program memory
- Immediate value → register
- Immediate value → data memory

Computational instructions
- Conditional ALU/MAC/SHIFT operation
- ALU/MAC/SHIFT operation with register ↔ register
- ALU/MAC/SHIFT operation with register ↔ data memory
- ALU/MAC/SHIFT operation with register ↔ program memory
- ALU/MAC operation with data memory → register and program memory → register

Program flow control
- Conditional jump
- Conditional subroutine call
- Conditional return
- Conditional trap
- Conditional do-until

Miscellaneous
- Saturate accumulator
- Modify index register
- Push status stack
- Pop status/loop/counter/PC stack
- Mode control
- No-op

Figure 5. ADSP-2100 instruction set summary.

This automatic looping mechanism eliminates the need for an explicit jump instruction within the loop. Every loop instruction is free to execute useful operations so the looping overhead is reduced to zero. Do-until loops can be any length, and the loop stack allows them to be nested four levels deep.

Instruction set

Figure 5 summarizes the ADSP-2100 instruction set. Four basic categories of instructions exist. The move instructions encompass register-register transfers, register-memory transfers, and immediate loading of registers and data memory. Data memory addresses are supplied either by the data address generators or directly from a field in the instruction word. Program memory addresses can only come from data address generator 2. Computational instructions exercise the ALU, MAC, and shifter functions. These functions can be executed conditionally based on current status register contents, or they can be combined with register-register and register memory move operations, including a simultaneous read of both program and data memories. The program-flow instructions direct the activities of the program sequencer. Execution of these instructions can either be unconditional or conditioned on the current status register contents. The miscellaneous instructions include saturation of the multiplier-accumulator output register, manual modification of address generator index registers, and manual pushing and popping of the various internal stacks. All of the instructions are coded into a single 24-bit word and execute in one cycle.

The 2100's instruction set with its diversity and parallelism does not lend itself to the usual mnemonic notation used for most computer assembly languages. Instead, the assembly syntax uses an algebraic notation that clearly spells out the action taken by each instruction. No user needs to memorize cryptic abbreviations to either write programs or read them. Straightforward notational conventions plus the general lack of arbitrary restrictions in the instruction set combine to produce assembly code that rivals high-level languages for ease of use and readability.

FIR filter

The features and capabilities of the 2100 are best demonstrated through programming examples of routines commonly used in DSP applications. A very basic but important DSP algorithm is the finite-impulse-response filter. A FIR filter directly implements a discrete convolution between a series of input samples and coefficients. Because of their simplicity and well-behaved numerical properties, FIR filters function in a wide range of problems, particularly in the area of telecommunications.[3]

See Figure 6 for a 2100 assembly-language subroutine for an FIR filter. Parameters pass to the routine through registers that are set up by the calling program. Index register I0 points to the filter delay line in data memory where the input samples are held. Index register I4 points to the filter coefficients that are stored in program memory. Modify registers M0 and M5, loaded with the value 1, move the delay line and coefficient pointers ahead one place each time they access memory. Finally, length register L0 is loaded with the filter order, which indicates the length of the delay line buffer. Each call to the subroutine takes one input sample from register AX0 and generates one output sample that is passed back to the calling program in register MR.

The actual filter code consists of only eight instructions. The first instruction (labeled FIRFILT) moves the contents of register L0 into register AY0. The assembler recognizes the equal sign as a transfer operator with the source on the right and the destination on the left. The second instruction combines an ALU operation with a data memory write. For the memory write, register AX0 provides the data, register I0 provides the memory address, and register M0 postmodifies I0. In other words, the input sample is written into the filter delay line and the pointer is incremented. The ALU portion of this instruction decrements register AY0 and places the result into register AR. AR now contains the filter order minus one, which is moved into the loop counter by the third instruction.

The fourth instruction does three things: It clears the MAC result register (MR), it fetches the first sample from

```
.MODULE FIR;
{
*********************************************************************************
*                                                                               *
*            FIR Filter Subroutine                                              *
*                                                                               *
*                                 L-1                                           *
*            Computation: Y(n) = SUM [H(k)*X(n-k)]                              *
*                                 k=0                                           *
*                                                                               *
*                            Y: output samples                                  *
*                            X: input samples                                   *
*                            H: coefficients                                    *
*                                                                               *
*            Input: I0 points to filter delay line (in data memory)             *
*                   I4 points to filter coefficients (in program memory)        *
*                   M0 contains 1                                               *
*                   M5 contains 1                                               *
*                   L0 contains filter order                                    *
*                   AX0 contains input sample                                   *
*                                                                               *
*            Output: MR contains output sample                                  *
*                                                                               *
*            Execution Cycles: L+9 (L = filter order)                           *
*                                                                               *
*********************************************************************************
}

FIRFILT:   AY0=L0;
           DM(I0,M0)=AX0, AR=AY0-1;                             { Store input sample }
           CNTR=AR;
           MR=0, MX0=DM(I0,M0), MY0=PM(I4,M5);                  { Clear Y, Get X, Get H }
           DO TAPLOOP UNTIL CE;
TAPLOOP:     MR=MR+MX0*MY0, MX0=DM(I0,M0), MY0=PM(I4,M5);       { Y=Y+(X*H), Get next X, Get next H }
           MR=MR+MX0*MY0 (RND);                                 { Y=Y+(X*H) }
           RTS;

.ENDMOD;
```

Figure 6. FIR filter subroutine.

data memory and places it into register MX0, and it fetches the first coefficient from program memory and places it into register MY0. Register I0 provides the address to the data memory and register I4 does the same for the program memory. M0 and M5 postmodify the two addresses. The fifth instruction sets up the looping hardware for a do-until loop. Taploop is a label given to the last instruction in the loop (in this case the only one), and CE refers to the counter-expired condition that terminates execution of the loop. The do-until instruction pushes the address of Taploop and the CE condition code onto the loop stack, and it pushes the contents of the program counter plus one onto the PC stack.

At this point the processor has internally stored all of the necessary information for sequencing through the loop. The beginning address of the loop is on the PC stack, and the ending address and termination condition are on the loop stack. The loop sequencing now becomes automatic and the loop instructions are not burdened by it. The FIR filter routine loop contains a single instruction that executes a multiply-accumulate operation and two memory fetches. The instruction syntax shows the MR register being loaded with the sum of itself and the product of registers MX0 and MY0. It also shows MX0 being loaded with the next sample from data memory while MY0 is loaded with the next coefficient from program memory. MX0 and MY0 supply the multiply operands before they are reloaded with new values. The instruction syntax clearly shows these actions when read from left to right. The processor will loop on this instruction, decrementing and testing the loop counter each time.

As the loop execution proceeds, the memory pointers increment through their respective buffers. The coefficient pointer I4 starts out at the beginning of the coefficient buffer and just reaches the end as the loop terminates. However this is not true for the sample delay line pointer I0. The filter delay line is a circular buffer whose starting point changes each time the filter routine is called. Since the samples do not physically move through the memory, the processor must realign its addressing each time a new output is computed. All of the delay line samples must be accessed on each pass, but the starting point is generally somewhere in the middle of the buffer. This means that a wraparound of the address must occur at some point. Putting the filter order (or equivalently the delay line length) into length register L0 takes care of this requirement automatically, using the modulo addressing capability.

It is important to note that the loop in this subroutine fits easily into the 2100's internal cache memory. So during loop execution, the cache can assume the program memory function, and the system performs like a three-memory architecture. All but one of the coefficients are fetched without incurring an extra cycle penalty. As with the modulo addressing, this action is transparent to the programmer. The processor fetches from cache whenever possible, without explicit codes or directions.

```
.MODULE BIQUAD;
{
*****************************************************************************
*                                                                           *
*       Biquad Cascade IIR Filter Subroutine                                *
*                                                                           *
*       Algorithm: For each biquad section                                  *
*                                                                           *
*                         2                    2                            *
*                Y(N) = SUM [B(K)*X(N-K)] + SUM [A(K)*Y(N-K)]               *
*                        K=0                  K=1                           *
*                                                                           *
*       Input: SR1 contains input sample                                    *
*              I0 points to delay line bufrer (in data memory)              *
*              I4 points to coefficient buffer (in program memory)          *
*              M0 contains 1                                                *
*              M1 contains -3                                               *
*              M4 contains 1                                                *
*              CNTR contains number of biquad sections                      *
*                                                                           *
*       Output: SR1 contains output sample                                  *
*                                                                           *
*       Cycles: (7*L)+10    where L is the number of biquad sections        *
*                                                                           *
*****************************************************************************
}

BIQUAD:       SE=SCALE;
              DO SECTIONS UNTIL CE;
                MX0=DM(I0,M0), MY0=PM(I4,M4);                    { GET X(N-2), GET B(2) }
                MR=MX0*MY0, MX1=DM(I0,M0), MY0=PM(I4,M4);        { X(N-2)*B(2), GET X(N-1), GET B(1) }
                MR=MR+MX1*MY0, MY0=PM(I4,M4);                    { X(N-1)*B(1), GET B(0) }
                MR=MR+SR1*MY0, MX0=DM(I0,M0), MY0=PM(I4,M4);     { X(N)*B(0), GET Y(N-2), GET A(2) }
                MR=MR+MX0*MY0, MX0=DM(I0,M1), MY0=PM(I4,M4);     { Y(N-2)*A(2), GET Y(N-1), GET A(1) }
                DM(I0,M0)=MX1, MR=MR+MX0*MY0 (RND);              { STORE X(N-1) AS X(N-2), Y(N-1)*A(1) }
SECTIONS:       DM(I0,M0)=SR1, SR=ASHIFT MR1 (HI);               { STORE X(N) AS X(N-1), ADJUST Y(N) }
              DM(I0,M0)=MX0;                                     { STORE Y(N-1) AS Y(N-2) }
              DM(I0,M0)=SR1;                                     { STORE Y(N) AS Y(N-1) }
              RTS;
.ENDMOD;
```

Figure 7. Biquad filter subroutine.

Loop execution terminates when the counter reaches zero. Automatic popping of the appropriate stacks restores the internal hardware to its original state, and program control transfers to the location immediately following the loop. These actions happen one cycle early, since the counter was loaded with the filter order minus one. The early transfer occurs because no additional memory fetches are needed after the last multiply-accumulate. This operation occurs outside the loop with the rounding option enabled so that the best 16-bit result can be obtained. The last instruction is a return-from-subroutine that transfers control back to the calling program.

Biquad IIR filter

Infinite impulse response (IIR) filters provide a somewhat different approach to the filtering problem. The basic computational operation is still multiply-accumulate, but feedback terms appear with the feed-forward paths, giving poles as well as zeroes. We refer to a second-order filter with two poles and two zeroes arranged so that all the products are accumulated into a single node as a biquad filter. Higher order filters can be constructed by cascading biquad sections.

Figure 7 displays the 2100 assembly code for a series of biquad filter sections. Input parameters to the subroutine pass through internal registers. The parameters include pointers to the filter delay line and coefficient buffer, several address modification values, and the number of cascaded filter sections. As with the FIR filter, each call to this subroutine accepts one input sample and generates one output sample.

Each biquad section theoretically requires two delay lines, one for the inputs (to feed forward) and one for the outputs (to feed back). However, since they are cascaded, the output delay line of the *n*th section is identical to the input delay line of the $n + 1$st section. There is no need to store redundant information, so the two delay lines are combined into a single one that is shared between the two adjacent sections. All of the combined delay lines are then concatenated to form a single buffer in memory.

The bulk of the subroutine consists of a do-until loop labeled Sections. The loop contains seven instructions and executes each biquad section in seven cycles. Most of the instructions perform an arithmetic operation with one or two memory accesses. As before, program memory stores the filter coefficients and data memory stores the delay line buffer, allowing simultaneous access. The modulo addressing feature is not used here at all. It is more desirable to have one index register access all of the concatenated delay lines than to dedicate a pointer to each one. Also, since the individual delay lines are only two samples long, not much effort is required to shift them manually. This step occurs as part of the last two instructions in the loop. The last instruction also performs an arithmetic scaling operation on the filter output. It corrects for any prescaling that may have been necessary to represent the coefficients in fixed-point format.

Fast Fourier transform

A somewhat more sophisticated example of the 2100's abilities is provided by that war-horse of DSP specsmanship, the fast Fourier transform. The importance of the FFT for this purpose is justified, for it taxes the computational and addressing capabilities of any processor. Its application to such areas as electronic instrumentation, radar systems, and speech processing gives it practical importance as well.[4]

The radix-2 FFT computation is divided into several stages. If the transform size is N (a power of two), there are Log(N) stages. All of the data samples are processed at each stage to produce new samples, which in turn are processed by the next stage. The butterfly, the basic kernel computation, operates on two complex samples at a time and generates two new samples. Each stage contains $N/2$ butterflies arranged in contiguous groups. For the decimation-in-frequency FFT the first stage has a single group of $N/2$ butterflies. In each successive stage the number of butterfly groups doubles and the size of each group is cut in half. In the last stage, there are $N/2$ groups with one butterfly each. The computation for each stage is usually performed "in-place" so that only one data storage area is required.

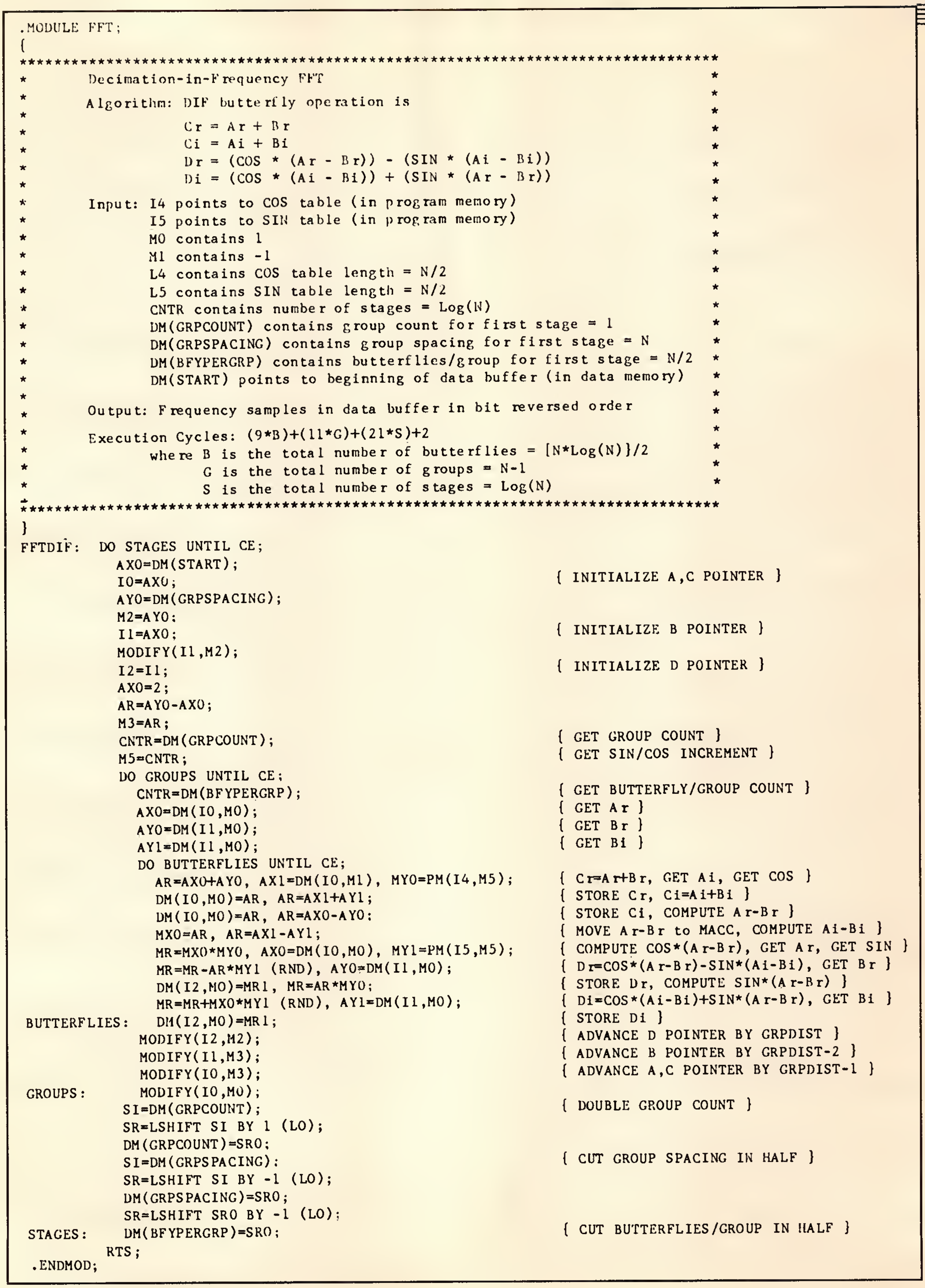
Figure 8 depicts a 2100 routine for a decimation-in-frequency FFT. The code has a structure that closely follows the format described above. Written as a generic subroutine, it transforms any power-of-two number of complex time samples into an equal number of complex frequency samples. Input parameters pass through registers and memory locations to define the size of the transform and the location of the data and coefficient buffers. Index registers I4 and I5 point to the cosine and sine tables in program memory. Modify registers M0 and M1 are set up to +1 and −1 so that the memory pointers can be moved both forward and backward. Length registers L4 and L5 contain the length information that controls the modulo addressing of the sine and cosine tables. The loop counter is loaded with the total number of stages in the transform. Three data memory locations store the initial values for the number of butterfly groups per stage, the memory spacing between adjacent groups, and the number of butterflies per group. One final data memory location provides the base address of the data buffer.

Although the code is a good deal more complicated than the filtering examples, the same basic mechanisms are at work. The three nested do-until loops correspond to the basic FFT entities, butterflies, groups, and stages. The innermost loop (butterflies) consists of nine instructions that execute the DIF butterfly computation. Packed into these nine instructions are eight arithmetic operations, 10 memory references, and a register move. Since this loop fits into cache memory, the extra cycles for the two program memory references disappear after the first pass. The butterfly loop is contained within another loop (groups) that executes a contiguous group of butterflies. The four instructions preceding the butterfly loop set up the counter and fill the data pipeline. Four instructions immediately follow the butterfly loop; they reposition the data memory pointers for the next group of contiguous butterflies. The outermost loop (stages) executes a complete pass through the data buffer, producing a new set of samples to be processed during the next pass. The instructions at the beginning of this loop initialize the various memory pointers and modify values for the first group of butterflies in the current stage. Instructions to update the parameters that define the grouping of butterflies for the next stage appear at the end of the loop.

The execution time for any size FFT can be computed easily. For instance, if $N = 1024$, the total number of butterflies is 5120, the total number of butterfly groups is 1023, and the number of stages is 10. Plugging these values into the execution cycle formula given in the subroutine header yields a total of 57,545 processor cycles. For a processor cycle time of 125 nsec, the 1024-point FFT executes in 7.2 msec.

The dynamic range of the FFT algorithm improves by maintaining a block-floating-point representation of the data. The 2100 FFT subroutine can easily be modified to accommodate a block representation by using the block exponent and normalization functions of the shifter. The block exponent derivation adds three cycles to the butterfly loop. This operation yields the exponent of the largest number produced at each stage. If this exponent is such that an overflow might occur during the next stage, every number in the data buffer is downshifted by an appropriate amount. This modification requires a two-cycle do-until loop at the end of the stage loop; the loop is executed only when downshifting is necessary. Assuming that downshifting occurs between half of the stages, a 1024-point, block floating-point FFT executes in only 11.7 msec. Table 1 summarizes the performance of the ADSP-2100 for these and other common algorithms.

The new generation of programmable DSP processors must be able to cope easily with such things as adaptive filtering, linear prediction, pattern recognition, dynamic programming, matrix decomposition, and vector quantization. Also an increasing need exists to deal with floating-point, complex, or multidimensional



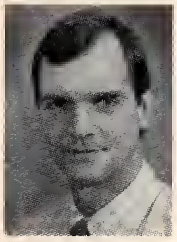
```
.MODULE FFT;
{
*****************************************************************************
*       Decimation-in-Frequency FFT                                         *
*                                                                           *
*       Algorithm: DIF butterfly operation is                               *
*                                                                           *
*                 Cr = Ar + Br                                              *
*                 Ci = Ai + Bi                                              *
*                 Dr = (COS * (Ar - Br)) - (SIN * (Ai - Bi))                *
*                 Di = (COS * (Ai - Bi)) + (SIN * (Ar - Br))                *
*                                                                           *
*       Input: I4 points to COS table (in program memory)                   *
*              I5 points to SIN table (in program memory)                   *
*              M0 contains 1                                                *
*              M1 contains -1                                               *
*              L4 contains COS table length = N/2                           *
*              L5 contains SIN table length = N/2                           *
*              CNTR contains number of stages = Log(N)                      *
*              DM(GRPCOUNT) contains group count for first stage = 1        *
*              DM(GRPSPACING) contains group spacing for first stage = N    *
*              DM(BFYPERGRP) contains butterflies/group for first stage = N/2 *
*              DM(START) points to beginning of data buffer (in data memory) *
*                                                                           *
*       Output: Frequency samples in data buffer in bit reversed order      *
*                                                                           *
*       Execution Cycles: (9*B)+(11*G)+(21*S)+2                             *
*              where B is the total number of butterflies = [N*Log(N)]/2    *
*                    G is the total number of groups = N-1                  *
*                    S is the total number of stages = Log(N)               *
*****************************************************************************
}
FFTDIF:  DO STAGES UNTIL CE;
            AX0=DM(START);
            I0=AX0;                                     { INITIALIZE A,C POINTER }
            AY0=DM(GRPSPACING);
            M2=AY0;
            I1=AX0;                                     { INITIALIZE B POINTER }
            MODIFY(I1,M2);
            I2=I1;                                      { INITIALIZE D POINTER }
            AX0=2;
            AR=AY0-AX0;
            M3=AR;
            CNTR=DM(GRPCOUNT);                          { GET GROUP COUNT }
            M5=CNTR;                                    { GET SIN/COS INCREMENT }
            DO GROUPS UNTIL CE;
              CNTR=DM(BFYPERGRP);                       { GET BUTTERFLY/GROUP COUNT }
              AX0=DM(I0,M0);                            { GET Ar }
              AY0=DM(I1,M0);                            { GET Br }
              AY1=DM(I1,M0);                            { GET Bi }
              DO BUTTERFLIES UNTIL CE;
                AR=AX0+AY0, AX1=DM(I0,M1), MY0=PM(I4,M5);   { Cr=Ar+Br, GET Ai, GET COS }
                DM(I0,M0)=AR, AR=AX1+AY1;                   { STORE Cr, Ci=Ai+Bi }
                DM(I0,M0)=AR, AR=AX0-AY0;                   { STORE Ci, COMPUTE Ar-Br }
                MX0=AR, AR=AX1-AY1;                         { MOVE Ar-Br to MACC, COMPUTE Ai-Bi }
                MR=MX0*MY0, AX0=DM(I0,M0), MY1=PM(I5,M5);   { COMPUTE COS*(Ar-Br), GET Ar, GET SIN }
                MR=MR-AR*MY1 (RND), AY0=DM(I1,M0);          { Dr=COS*(Ar-Br)-SIN*(Ai-Bi), GET Br }
                DM(I2,M0)=MR1, MR=AR*MY0;                   { STORE Dr, COMPUTE SIN*(Ar-Br) }
                MR=MR+MX0*MY1 (RND), AY1=DM(I1,M0);         { Di=COS*(Ai-Bi)+SIN*(Ar-Br), GET Bi }
 BUTTERFLIES:   DM(I2,M0)=MR1;                              { STORE Di }
              MODIFY(I2,M2);                                { ADVANCE D POINTER BY GRPDIST }
              MODIFY(I1,M3);                                { ADVANCE B POINTER BY GRPDIST-2 }
              MODIFY(I0,M3);                                { ADVANCE A,C POINTER BY GRPDIST-1 }
 GROUPS:      MODIFY(I0,M0);
            SI=DM(GRPCOUNT);                                { DOUBLE GROUP COUNT }
            SR=LSHIFT SI BY 1 (LO);
            DM(GRPCOUNT)=SR0;
            SI=DM(GRPSPACING);                              { CUT GROUP SPACING IN HALF }
            SR=LSHIFT SI BY -1 (LO);
            DM(GRPSPACING)=SR0;
            SR=LSHIFT SR0 BY -1 (LO);
 STAGES:    DM(BFYPERGRP)=SR0;                              { CUT BUTTERFLIES/GROUP IN HALF }
          RTS;
 .ENDMOD;
```

Figure 8. FFT subroutine.

data. The ADSP-2100 expands the realm of programmable signal processors into these areas, making possible the implementation of DSP systems that previously would have required special-purpose solutions.[5]

References

1. J. P. Roesgen, "A High Performance Microprocessor for DSP Applications," *Proc. ICASSP 86,* Vol. 1, pp. 397-400.
2. D. Bursky, "Digital Signal Processing Chips Move Off Designers's Wish List and Into Everyday Use," *Electronic Design*, Vol. 32, No. 10, pp. 100-122.
3. J. P. Roesgen, "Fast Modem Designs Benefit From DSP Chip's Versatility," *Electronic Design*, Vol. 34, No. 13, pp. 123-128.
4. A. V. Oppenheim and R. W. Schafer, "Digital Signal Processing," Prentice-Hall, Englewood Cliffs, N.J., 1975.
5. J. Allen, "Computer Architecture for Digital Signal Processing," *Proc. IEEE*, Vol. 73, No. 5, pp. 852-873.



John P. Roesgen is a senior member of the technical staff at Concord Data Systems, Marlboro, Massachusetts. While at Analog Devices, Inc., he was responsible for the architecture definition and logic design of the ADSP-2100 processor. His areas of interest include digital signal processing, communication systems, and VLSI design.

Roesgen holds a BSEE degree from the University of Connecticut and an MSEE degree from Worcester Polytechnic Institute. He is a member of Tau Beta Pi, Eta Kappa Nu, and the IEEE.

Questions about this article can be directed to Roesgen at Concord Data Systems, 397 Williams St., Marlboro, MA, 01752. Direct questions about the ADSP-2100 microprocessor to Analog Devices, Inc., DSP Division, PO Box 280, Norwood, MA 02062.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 180 Medium 181 Low 182

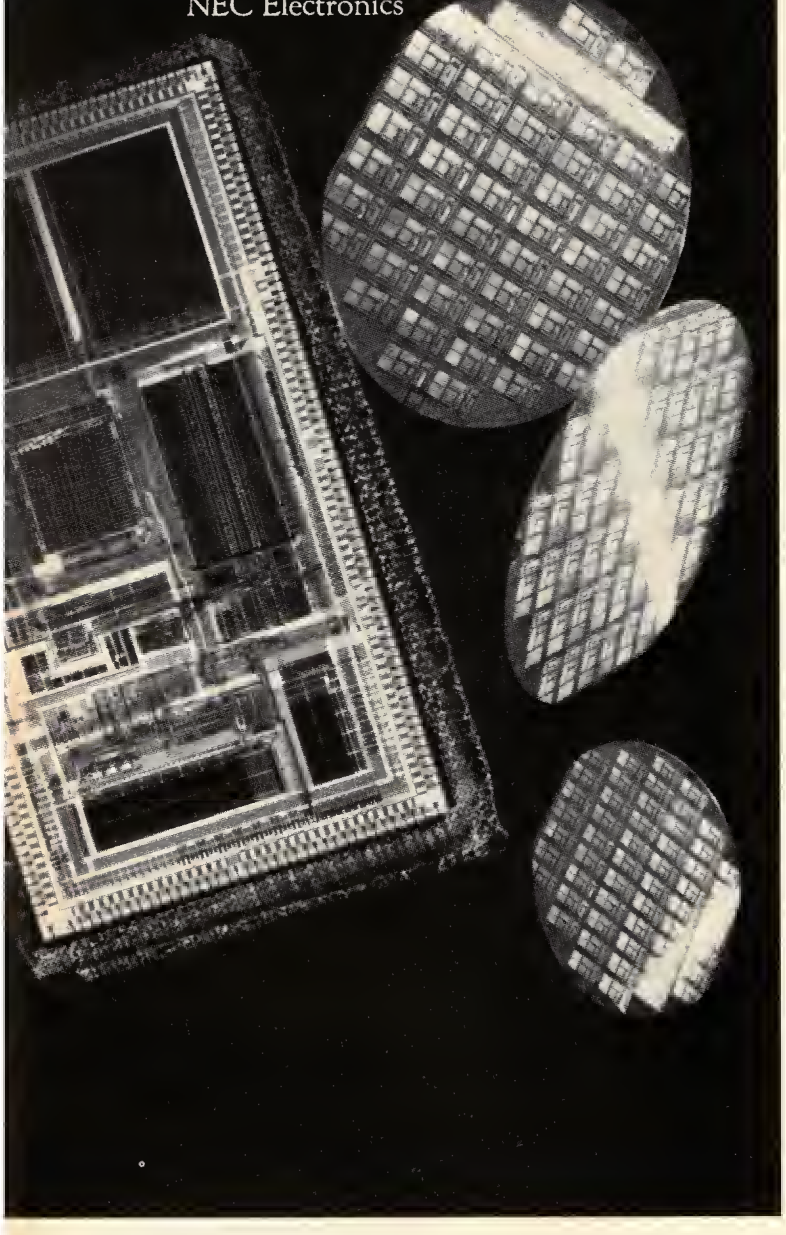
Table 1.
ADSP-2100 performance figures.

Routine	Execution time
64-tap FIR filter	8.0μs per output sample
64-tap, complex FIR filter	32.0μs per output sample
Biquad filter section	0.88μs per section
Normalized lattice filter section	0.63μs per section
1024-point, complex FFT	7.2ms total
64-tap FIR filter gradient adaptation	16.0μs total
Two-dimensional convolution (3 × 3 mask)	2.5μs per output sample
Matrix multiply (10 × 10 matrices)	0.22ms total
Floating-point multiply-accumulate	1.625μs total
Trigonometric sine	3.25μs total

This 150-ns device performs full 32-bit floating-point arithmetic, incorporates on-chip instruction and data memory, and provides both serial and parallel I/O.

NEC's μPD77230 Digital Signal Processor

Bill Eichen
NEC Electronics



Over the past several years, microprocessor technology has produced a number of interesting offshoots, including single-chip digital signal processors. The primary feature that distinguishes a digital signal processing device from an "ordinary" microprocessor is an on-chip hardware multiplier that operates in a single instruction cycle. Typically, a large number of multiplications are employed in DSP algorithms, and the kernels of such algorithms consist of a few operations repeated many times. These algorithms often implement a sum of products, in which many multiplications must be done on each input sample. These algorithms are used in applications such as finite impulse response (FIR) filters, infinite impulse response (IIR) filters, autocorrelators, and fast Fourier transforms (FFTs). Multiplication speed is therefore one of the most important considerations in choosing a DSP device for a given application.

Here, we present a brief history of DSP devices, the rationale for developing a floating-point signal processor, details of the hardware architecture of a new digital signal processor from NEC Electronics, an overview of its instruction set, and certain benchmarks and code examples.

DSP history

The Intel 2920, introduced in 1980, marked the first generation of single-chip DSP devices. Although it did not include a hardware multiplier, it may be considered a DSP device since it contained on-chip A/D and D/A converters and was designed for small, repetitive tasks.

The second generation of single-chip DSP devices is typified by the NEC μPD7720 and the Texas Instruments TMS320. Both provide a one-instruction-cycle multiplier, but they have somewhat different architectural philosophies. The TMS32010, which was introduced in 1982, has an instruction set similar to that of a general-purpose microprocessor—it has instructions like LOAD, MOVE, and ADD. The NEC μPD7720, which was introduced in 1980, has a microcode-like instruction set that it combines with a parallel architecture to enable a single instruction to load the two multiplier inputs, accumulate the multiplied product, modify both RAM/ROM pointers, and execute a return from subroutine.

When faced with an application whose requirements exceed the capability of a single-chip DSP device such as the

0272-1732/86/1200-0060$01.00 © 1986 IEEE

TMS32010 or μPD7720, many system designers turn to a bit-slice solution. Bit-slice architectures make use of independent data paths and parallel structures in which microcoding is employed. Although bit-slice processors provide very high performance, they are difficult to program and frequently require hardware to be reconfigured. They also must be built out of several discrete components.

Now, third-generation single-chip DSP devices, which are characterized by fabrication in low-power CMOS, fast instruction cycle times, high-precision arithmetic, and on-chip resources such as RAM and ROM, are becoming available. Their high performance, low cost, and ease of use make them an attractive replacement for bit-slice processors.

The NEC μPD77230

NEC Electronics recently introduced the first member of a family of third-generation, CMOS digital signal processors. Called the μPD77230 Advanced Signal Processor, or ASP, the new device incorporates full 32-bit floating-point (24-bit mantissa, 8-bit exponent) arithmetic, a 150-ns instruction cycle time (even for multiply/accumulate), a 1K × 32-bit internal RAM, a 2K × 32-bit internal instruction ROM, a 1K × 32-bit data ROM, and serial and parallel I/O. The device integrates more than 370,000 transistors in a 1.75-μm CMOS process and dissipates less than one watt. It is packaged in a 68-pin grid array.

A variant of the μPD77230 ASP will be introduced—the μPD77220, a fixed-point-only version of the μPD77230, will have half its internal RAM (2 × 256 × 24), will cost less, and will consume approximately 0.7 watt.

The rationale for a floating-point digital signal processor

Many first-generation DSP chips represent numbers in fixed-point form. Samples are therefore peak-limited by $+1$ and -1, sometimes requiring that the filter coefficient values be prescaled. This prescaling operation frequently introduces a round-off error into the system. For this reason, finite-length integer calculations are often modeled with a "white noise" error source added to the incoming signal. Not only can truncation and rounding off cause a loss of precision, but the accumulated products of truncated or rounded values can alter the system's overall transfer function. The relocating of these poles and zeroes can introduce instabilities into the system. Furthermore, fixed-point calculations are very susceptible to overflow (underflow) because of their limited word length. Many fixed-point systems require overflow checking routines that saturate overflow values. Not only is a significant level of error introduced when a value is clamped, but additional programming overhead is required. When efficient code for sum-of-product (finite impulse response) filters is being developed, for example, overflow detection—when required—can account for a substantial number of instructions. Multiple-stage operations such as the biquad filter also require overhead for overflow checking.

Floating-point devices avoid many of these drawbacks. The ASP, for example, provides 24-bit precision in the mantissa, just as a 24-bit fixed-point device does, but also employs an 8-bit, two's-complement exponent, which increases the dynamic range of the signal. Since the ASP can represent much smaller numbers, it enhances the overall precision of the system in which it is used. In most cases, overflow checking is unnecessary with the ASP. Eight bits of exponent permits numbers with absolute values as large as 1.7×10^{38} and as small as 3.5×10^{-46}.

ASP hardware architecture

The architecture of the μPD77230 ASP is quite similar to that of a microcoded building block system. It incorporates a full 32 × 32-bit floating-point multiplier, a 55-bit floating-point ALU, a 47-bit barrel shifter, a program and data ROM, a data RAM, and control (Figure 1). By interconnecting these functional blocks with multiple data paths on a single chip, the ASP allows several operations to occur simultaneously. For example, the ASP can perform a floating-point multiply, a floating-point addition, dual base and index pointer moves, a barrel shift, an internal data bus register transfer, and serial I/O in a single 150-ns cycle. In contrast, many other digital signal processors are restricted to operating on only one functional block (the multiplier, for example) in a cycle. The ASP therefore combines the flexibility of a general-purpose microprocessor with the computational power of high-performance signal processing elements.

The ASP's processing unit consists of three major blocks: the ALU, the barrel shifter, and the working registers (accumulators). The ALU is a 47-bit, fixed-point unit using two inputs selected by two input registers, P and Q. The Q register selects input from one of the eight working registers, while the P register chooses input from one of four sources—the main bus, the multiplier output, or one or the other of the two internal RAM blocks. The ASP employs a microcoded type of instruction set that multiplexes the P and Q inputs to the ALU. The ALU can perform a 55-bit floating-point add/subtract, a 47-bit fixed-point add/subtract, and 47-bit logical operations (XOR, AND, OR, and NOT). It can also reverse the bit order to aid in addressing an in-place fast Fourier transform operation.

The processing unit uses the 47-bit bidirectional barrel shifter in several ways. During a floating-point add or subtract, for example, the exponent arithmetic unit, or EAU, signals the barrel shifter to align the two floating-point numbers so it can add their mantissas. And when fixed-point arithmetic is being performed, a shift value may be specified in the same instruction. This value is latched into the shift value register, or SVR, so that it need not be specified on each instruction. The SVR shifts the P input before it enters the ALU, and therefore the P input can be used to prescale a series of numbers.

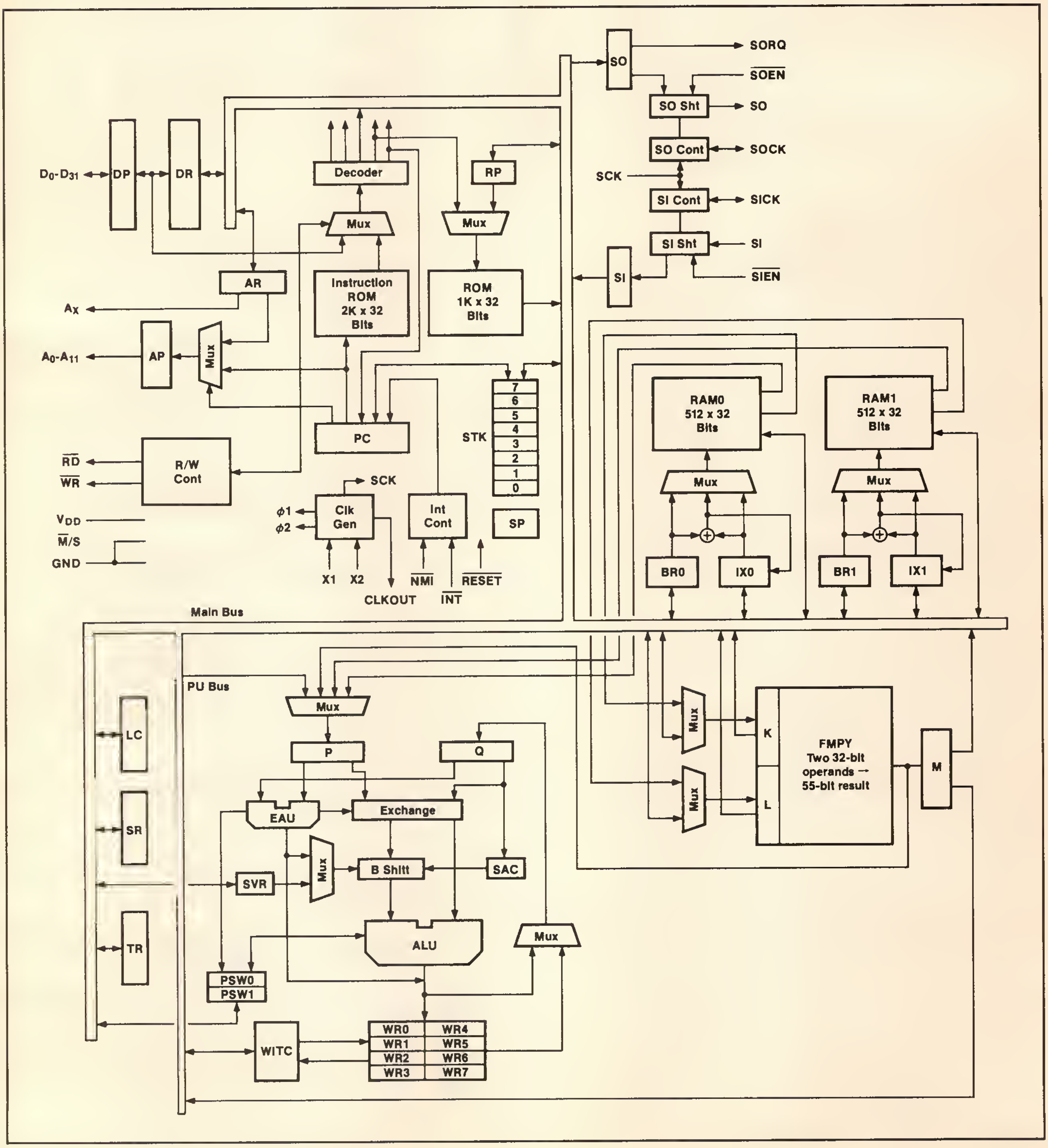


Figure 1. Block diagram of the μPD77230.

The barrel shifter can also be used to shift a working register by *n* bits, either left or right.

Finally, the barrel shifter is used whenever a normalization instruction is executed in one of the working registers. This occurs during conversions between ASP and IEEE 32-bit floating-point formats. As we noted before, the ASP floating-point format uses a two's-complement exponent and mantissa. The ASP's designers chose this format to reduce hardware complexity. The IEEE format, however, has an offset exponent and sign-magnitude mantissa. In order to convert numbers to IEEE format, the ASP must add the offset to the exponent, change the mantissa to an absolute value, and then rotate and fill the mantissa to yield the proper hidden bit format.

There are eight 55-bit working registers on the processing unit bus; they can be exchanged or moved via the main bus into any other register or RAM location. For example, when a finite impulse response, or FIR, filter is being performed, a working register can be continuously accumulated—that is, it can be fed back to the floating-point adder on each successive cycle. This implies that FIR filters require only one instruction cycle per tap.

The multiplier section has two 32-bit floating-point inputs that can be loaded simultaneously, one from the main bus and the other from a special sub-bus that accesses either of the two independent RAM blocks. Internally, the multiplier consists of a 24 × 24-bit, two's-complement, fixed-point multiplier with a 47-bit result and an 8-bit exponent adder that yield a 55-bit floating-point product. This product can be routed to a number of destinations, input to the floating-point ALU, moved to one of the working registers, or truncated to 32 bits and stored in other registers or internal RAM. The multiplier is implemented through a modified Booth's algorithm in a Wallace tree configuration.

ASP memory and addressing

The μPD77230 ASP features 1K × 32 bits of internal RAM, 2K × 32 bits of internal instruction ROM, and 1K × 32 bits of internal data/coefficient ROM. Both ROM areas can be masked. The ASP employs several memory addressing modes, each optimized for particular DSP operations.

The 1K × 32 bits of data RAM are organized as two separately addressable 512 × 32-bit blocks. Each block has its own base and index pointer that can be independently modified in an instruction. Furthermore, the base registers can be placed in a modulo count mode. In this mode, the upper *n* bits ($n = 1, 2, 3, \ldots, 8$) are set to a fixed pattern and the lower $(9 - n)$ bits can be used to cycle through a table and wrap around to zero without generating a carry to the higher *n* bits. The RAM blocks are available to the main bus, the P input register, or either of the multiplier inputs.

The 1K × 32 bits of data/coefficient ROM are accessed either by the 10-bit ROM pointer (which is incremented or decremented) or by the 9-bit field immediately specified by an instruction. The ROM includes a special modification feature that allows a 2^n add operation. ROM output data are available for use anywhere in the device via the main bus.

The 2K × 32 bits of internal instruction ROM are controlled by a 13-bit program counter that is saved on an eight-level internal stack during subroutine and interrupt calls. The contents of the top of the stack are available to the main bus, and consequently the programmer can directly modify the program counter.

ASP operating modes

The ASP is configured at power-up to perform in either a master or slave mode. In master mode, it connects 32 data pins to the system data bus. It directly addresses up to 4K words of external instructions and up to 8K words of external data. The external memory overlaps in such a way that if 4K of external instruction is used, only 4K of external data is available. Furthermore, the lower 4K of external memory is high-speed and can be accessed in a single cycle. This implies that the external instruction memory in the high-speed area must have an access time of 45 ns. There is no loss in speed if the ASP uses an external instruction instead of one from internal memory. The low-speed area stores data only and accesses standard 250-ns memory in three cycles.

In slave mode, the ASP interfaces to the system bus through a 16-bit I/O port that transfers data in 16- or 32-bit words. A local data bus, accessible only to a slave ASP, has a data length that can be programmed, in byte increments, to be from 8 to 32 bits. In addition, a slave ASP provides four general-purpose I/O pins: two programmable outputs and two testable inputs.

The ASP includes a status register for controlling the various operating modes. It also provides both maskable and nonmaskable interrupt control and incorporates a loop counter register that is automatically decremented. It has an external system clock that can be used to synchronize multiprocessor configurations.

The ASP's independent serial I/O sections are ideal for interfacing it directly to a codec or a successive-approximation analog-to-digital converter. The ASP's serial interface section contains serial shift registers, parallel-to-serial converters, and input/output control circuitry. The serial output of one ASP can be directly cascaded to the serial input of another ASP. The serial input and output sections are independently programmable—the transfer-bit length for each can be specified, in byte increments, to be from 8 to 32 bits. They can also be configured to use an internal or external serial clock, either allowing a transfer rate of up to 5 MHz. The serial input and output sections also have separate enable signals and they automatically reset during loss of synchronization.

The ASP instruction set

The instruction set of the μPD77230 ASP is best described as horizontal microcode. There are three basic types of instructions: OP, load immediate, and branch (see Figure 2). The highly parallel architecture of the ASP enables a single instruction to perform several simultaneous operations. All instructions execute in a single 150-ns cycle and occupy one word (32 bits) of instruction memory. As mentioned earlier, there is no penalty in speed associated with the use of the external instruction space.

The most frequently used instruction type is OP. OP instructions have six separate fields of microcoded bits that perform the following functions: select ALU operation, select ALU operands, transfer internal data (including the double load of multiplier inputs), modify RAM pointers, modify ROM pointers, and control modes.

There are 26 ALU operations that process two 32-bit floating-point numbers to produce a 55-bit result (Table 1). The P and Q inputs select the operands for the ALU operation. The Q input (three bits) selects one of the eight working registers (accumulators), while the P input (two bits)

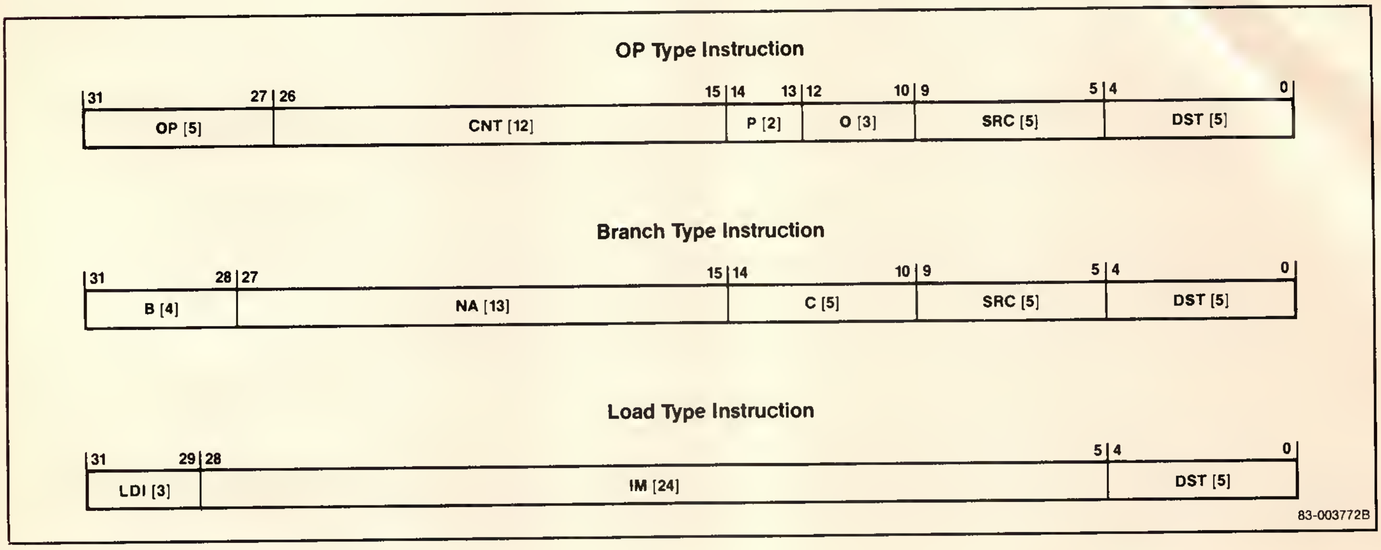


Figure 2. μPD77230 instruction types.

Table 1.
Specifications for the fields in OP instructions.

Mnemonic	Operation
NOP	No operation
INC	Increment
DEC	Decrement
ABS	Absolute value
NOT	Not—ones complement
NEG	Negate—twos complement
SHLC	Shift left with carry
SHRC	Shift right with carry
ROL	Rotate left
ROR	Rotate right
SHLM	Shift left multiple
SHRM	Shift right multiple
SHRAM	Shift right arithmetic multiple
CLR	Clear
NORM	Normalize
CVT	Convert floating-point format
ADD	Fixed-point add
SUB	Fixed-point subtract
ADDC	Fixed-point add with carry
SUBC	Fixed-point subtract with borrow
CMP	Compare (floating point)
AND	Logical AND
OR	Logical OR
XOR	Logical exclusive OR
ADDF	Floating-point add
SUBF	Floating-point subtract

selects either the main bus, multiplier output, or one of the two RAM blocks.

The internal data transfer field moves one of 32 source and destination registers in parallel with the execution of an OP instruction. The multiplier is always active and generates a product on each cycle. If new inputs are not loaded during an instruction, the previous values are simply remultiplied.

The control field can assume one of 15 combinations of subfields. These subfields are specified in an instruction by the appropriate mnemonic. The assembler configures the operation and checks for legal combinations of specifiers. The subfields control RAM operating modes, ROM base/index pointers, loop counter decrementing, value/normalization register shifting, transfer formatting, and the like.

A "load immediate data" instruction specifies the 24-bit mantissa and its destination register. Since an instruction word is only 32 bits wide, floating-point data must be loaded in two cycles. In this case, the eight-bit exponent is stored beforehand in a temporary register.

The branch instruction type includes jump, conditional branch, subroutine call, and return instructions. Branch instructions employ a 5-bit condition field and a 13-bit next-address field. A branch instruction can perform an internal data bus transfer while it is executing regardless of the branch condition. Because of the pipelining of the ASP, the instruction following a branch instruction is always executed.

In general, the more pipelined a processor's architecture, the more difficult it becomes to branch and clear the contents of the pipeline. The ASP uses a relatively simple three-stage pipeline—an instruction fetch, execution, and result occur in three successive cycles pipelined to yield one-cycle-per-instruction throughput. The latency is three cycles, but this is usually insignificant since many DSP programs are long and repetitive.

A side effect of this pipelined operation is that the instruction immediately following a branch is always executed, regardless of whether the branch condition was met. This occurs because the pipeline has already prefetched the instruction following the branch. One consequence of this side effect is that the programmer will make a branch the next-to-last instruction in a loop. In this way, he ensures that the last instruction of the loop will always be executed. An example of this pipelining can be seen in the code for a FIR filter (Figure 3).

```
Initial Conditions:
Loop Counter (LC) = Number of taps
RAM0 contains the delay taps X(n−i)
RAM1 contains the coefficients A(i)
Base Pointer 0 (BP0) = First delay tap
Base Pointer 1 (BP1) = First coefficient
Working Register 0 (WR0) = New input sample
--------------------------------------------------------
Note: Instructions are separated by horizontal
      lines. Multiple entries represent subfields of
      a single instruction.
--------------------------------------------------------
MOV RAM0, WR0       Input sample stored in RAM0
CLR WR0 ;           Clear working register for
                    summation output
--------------------------------------------------------
MOV KLR1, RAM0 ;    Load multiplier with first
                    sample and coefficient
--------------------------------------------------------
MOV TR, K           Save input sample in
                    temporary register
INCBP0              Move Base Pointer 0 to first
                    delay tap
INCBP1 ;            Move Base Pointer 1 to
                    second coefficient
--------------------------------------------------------
START:              Beginning of loop
ADDF WR0, M         Add previous multiplier
                    result to summation
MOV KLR1, RAM0 ;    Load next delay tap and
                    coefficient to multiplier
--------------------------------------------------------
MOV RAM0, TR        Save previous delayed sample
                    in current tap
DECLC ;             Decrement Loop Counter,
                    zero skips next instruction
--------------------------------------------------------
JMP START ;         If Loop Counter not zero,
                    loop back for next tap
--------------------------------------------------------
                    The next instruction will
                    always be executed even
                    during branch, due to pipe-
                    lining
MOV TR, K           Retrieve current delay tap
                    from multiplier reg
INCBP0              Move pointer to next delay
                    tap
INCBP1 ;            Move pointer to next co-
                    efficient
--------------------------------------------------------
```

Figure 3. Code for the FIR filter. Example 1—repetitive loop calculation.

Code examples

Below, we describe in detail the implementation of some frequently used DSP operations. We explain the algorithms for the finite impulse response (FIR) filter, the biquad filter, and the fast Fourier transform (FFT) and show μPD77230 source code for each. In each case, the algorithms are implemented for floating-point operations. However, with only slight modification they can be performed at the same speed in a fixed-point environment. (See benchmarks in Table 2.)

Table 2.
μPD77230 benchmarks.

Division	4.8 μs
Square root (Newton's method)	6.0 μs
SIN, COS (Taylor series)	10.8 μs
ATAN (Maclaurin expansion)	40.0 μs
Biquad filter (1 stage)	0.9 μs
FIR filter (32 taps)	5.2 μs
Complex FFT	
32 points	150.0 μs
512 points	4.7 ms
1024 points	12.5 ms
	(Uses external memory)

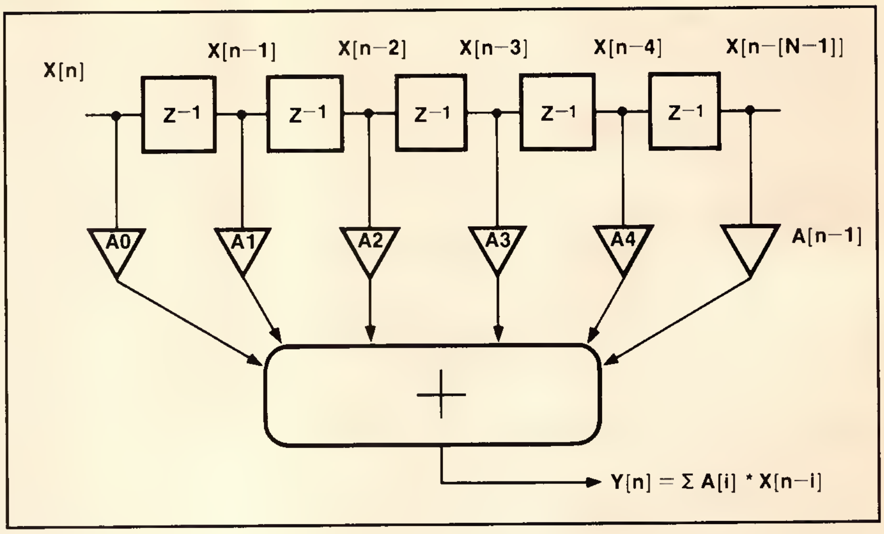


Figure 4. Flow diagram for the FIR filter.

Finite impulse response filter. The FIR filter is simply a sum of products with no feedback terms. The output of each sample is a weighted sum of the new input value and the $n-1$ previous delayed samples (Figure 4). The weighted coefficient values are selected to produce a given frequency response for the digital filter.

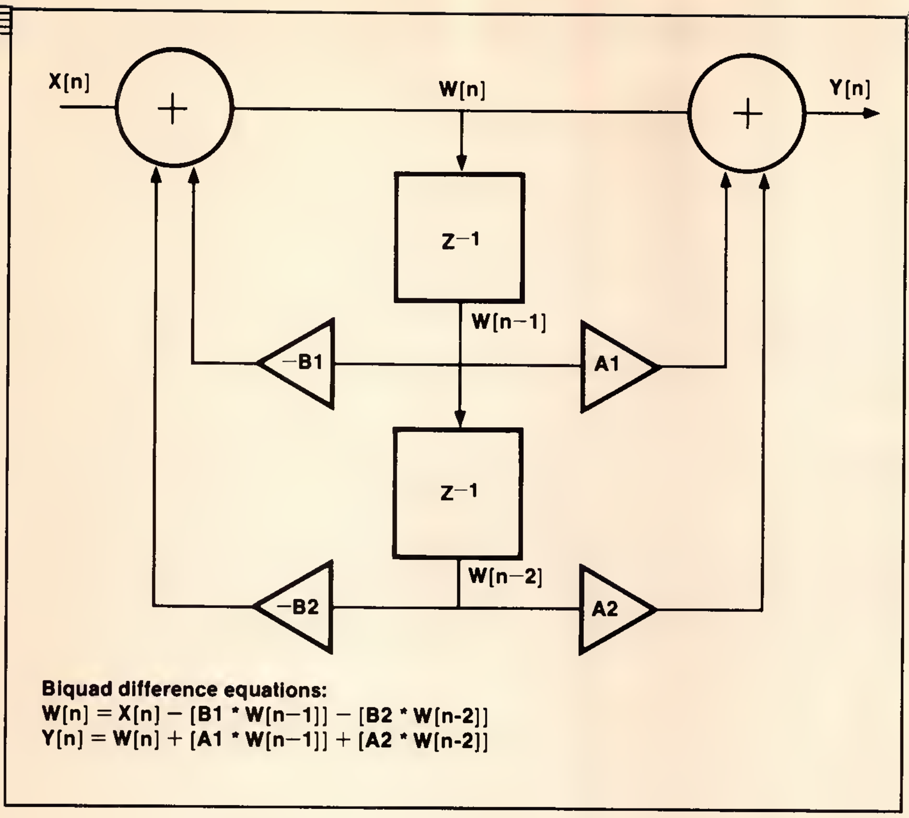
Speed and program space can be traded off in the implementation of an FIR filter. In one algorithm, the basic loop moves the nth sample and the nth coefficient to the multiplier inputs, accumulates the previous multiplier result, replaces the $n-1$th delay tap with the nth tap, decrements a counter, and then branches to the beginning. Although this implementation of an FIR filter is extremely inefficient (it requires four cycles per tap), it works in the general case. Figure 3 shows the code for this implementation.

When n is a power of two, however, the ASP's modulo count capabilities can be employed. A circular buffer can be implemented that simulates replacing the $n-1$th tap with the nth tap by restoring the pointer after the entire filter length has been executed. Instead of physically saving the

```
Initial Conditions:
Number of taps (N) must be a power of 2
Modulo Counter is set to log2(N)
RAM0 contains the delay taps X(n−i)
RAM1 contains the coefficients A(i)
Base Pointer 0 (BP0) = First delay tap
Base Pointer 1 (BP1) = First coefficient
Working Register 0 (WR0) = New input sample
------------------------------------------------------
Note: Instructions are separated by horizontal
      lines. Multiple entries represent subfields of
      a single instruction.
------------------------------------------------------
MOV RAM0, WR0      Place input sample in RAM0
CLR WR0 ;          Clear Working Register 0 for
                   summation
------------------------------------------------------
MOV KLR1, RAM0     Load multiplier with input
                   sample and coefficient
INCBP0             Move Base Pointer 0 to first
                   delay tap
INCBP1 ;           Move Base Pointer 1 to
                   second coefficient
------------------------------------------------------
```

```
     Next instruction is repeated N−2 times,
             N = Number of taps
------------------------------------------------------
ADDF WR0, M        Add previous multiplier
                   result to summation
MOV KLR1, RAM0     Load next delay tap and
                   coefficient to multiplier
INCBP0             Move pointer to next delay
                   tap
INCBP1 ;           Move pointer to next co-
                   efficient
------------------------------------------------------
         Repeat above instruction N−2 times
------------------------------------------------------
ADDF WR0, M        Add (N−1)th product, end
                   condition
MOV KLR1, RAM0     Load last sample and
                   coefficient to multiplier
INCBP1 ;           Move pointer to first
                   coefficient (modulo wrap)
                   but do not increment delay
                   tap pointer so that circular
                   buffer can be implemented
------------------------------------------------------
ADDF WR0, M ;      Add final product to
                   accumulated sum in WR0,
                   next time through, taps will
                   delay by one
------------------------------------------------------
```

Figure 5. Code for the FIR filter. Example 2—straight in-line code.



Biquad difference equations:
$W[n] = X[n] - [B1 * W[n-1]] - [B2 * W[n-2]]$
$Y[n] = W[n] + [A1 * W[n-1]] + [A2 * W[n-2]]$

Figure 6. Flow diagram for the biquad filter.

delayed samples using a temporary register, the RAM pointers cycle through the table. The modulo count causes a wrap-around, eliminating the need to test for the end of the buffer.

If an application requires real-time speed, straight in-line coding can be employed. This way of structuring a program is very similar to a circular buffer except that it does not use a loop counter and branch instruction, which are unnecessary. Figure 5 shows the straight in-line code for the FIR filter.

Biquad filter. One of the most common operations for a digital signal processor is the biquadratic filter. The biquad filter is a two-tap FIR filter with feedback. The signal flow diagram for this filter (Figure 6) shows that for each new input sample, four multiplications, four additions, and two delays are needed to generate each output value. The code for the biquad filter is shown in Figure 7. The ASP can perform this filter in six cycles, or 0.9 μs. Note that the ASP processes fixed-point and floating-point numbers with the same speed. However, floating point avoids the need for overflow detection coding, which can slow the system.

The initial conditions for biquad filtering require the ROM pointer (RP) to be set at the top of the coefficient table, the RAM0 base pointer (BP0) to be set to the first delay tap, and the new input sample to reside in working register 0 (WR0). The result will be placed in WR1. One can cascade biquad sections by either adding an instruction to move the result back to WR0 or copying the biquad filtering routine and reversing WR0 and WR1.

The biquad algorithm can be rewritten to use RAM1 instead of the fixed internal data ROM. If this is done, the

```
Note: Instructions are separated by horizontal
      lines. Multiple entries represent subfields of
      a single instruction.
-------------------------------------------------------
MOV LKR0, ROM     K = W(n-1), L = -B1 (Load
                  Multiplier)
CLR WR1           Clear WR1 for summation
DECRP             RP points to -B2
INCBP0 ;          Base Pointer moves to
                  W(n-2)
-------------------------------------------------------
ADDF WR0, M       WR0 = X(n) - (B1 * W(n-1))
MOV LKR0, ROM     K = W(n-2), L = -B2
DECRP ;           RP moves to A2 (BP0 still at
                  W(n-2))
-------------------------------------------------------
ADDF WR0, M       WR0 = X(n) - (B1 * W(n-1))
                  - (B2 * W(n-2)) = W(n) =
                  New W(n-1)
MOV LKR0, ROM     K = W(n-2), L = A2
DECRP             RP moves to A1
DECBP0 ;          BP0 moves to W(n-1)
-------------------------------------------------------
ADDF WR1, M       WR1 = 0 + (A2 * W(n-2))
MOV LKR0, ROM ;   K = W(n-1), L = A1
-------------------------------------------------------
ADDF WR1, IB      WR1 = WR0 (New W(n-1))
                  + (A2 * W(n-2))
MOV RAM0, WR0     Save New W(n-1) in place in
                  RAM
DECRP             RP points to top of next
                  stage's table
INCBP0 ;          BP0 moves to W(n-2)
-------------------------------------------------------
ADDF WR1, M       WR1 = New W(n-1) + (A2 *
                  W(n-2)) + (A1 * Old
                  W(n-1))
MOV RAM0, K       Old W(n-1) stored to
                  W(n-2)
INCBP0 ;          BP0 moves to W(n-1) for
                  next stage
-------------------------------------------------------
```

Figure 7. Code for the biquad filter.

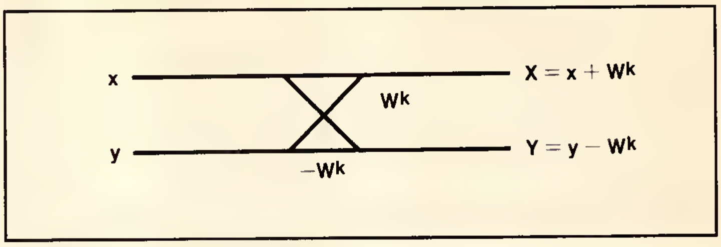


Figure 8. Flow diagram for the FFT butterfly.

coefficients can be updated for applications such as adaptive filtering and equalization.

Fast Fourier transform. Another common DSP operation is the fast Fourier transform, or FFT, which is a mathematically efficient method for performing the discrete Fourier transform. The basic operation of the FFT is the "butterfly," which consists of a two-point FFT (Figure 8). Here, the input data (x,y) and output data (X,Y) are complex. The FFT algorithm takes advantage of the base and index pointers of the ASP, using the index register as an offset to simplify addressing of the butterfly. It sets the base pointer to the first value, x, and the index pointer to the offset of the second value, y. Note that because the algorithm employs decima-

```
Initial Conditions:
RAM0 contains real part of input data
RAM1 contains imaginary part of input data
Base Pointer 0 (BP0) = 1H (Hex)
Base Pointer 1 (BP1) = 1H
Index Register 0 (IR0) = 10H
Index Register 1 (IR1) = 10H
ROM Pointer (RP) = 2H
Special ROM Pointer Increment = 2 (used with
INCBRP mnemonic)
-----------------------------------------------------------
CLR WR0              Working Register 0 (WR0)
                     clear
MOV LKR0, ROM        K ← y_r    L ← cos(z)
RPINC                ROM Pointer = 3H
SPCBI1 ;             RAM1 uses Base1 + Index1
                     (RAM1 = 11H)
-----------------------------------------------------------
ADDF WR0, M          WR0 = cos(z) * y_r
MOV KLR1, ROM        K ← sin(z)    L ← y_i
SPCBP0 ;             RAM0 uses Base0 (RAM0 =
                     1H)
-----------------------------------------------------------
ADDF WR0, M          WR0 = cos(z) * y_r + sin(z) *
                     y_i
MOV WR1, RAM0        WR1 ← x_r
SPCBP1 ;             RAM1 uses Base1 (RAM1 =
                     1H)
-----------------------------------------------------------
ADDF WR0, RAM0       WR0 = x_r + cos(z) * y_r +
                     sin(z) * y_i
MOV WR3, RAM1 ;      WR3 ← x_i
-----------------------------------------------------------
SUBF WR1, IB         WR1 = x_r − (cos(z) * y_r +
                     sin(z) * y_i)
MOV NON, WR0         Use value on internal data
                     bus (IB)
SPCBI0 ;             RAM0 uses Base0 + Index 0
                     (RAM0 = 11H)
-----------------------------------------------------------
CLR WR2              Clear Working Register 2
MOV LKR0, ROM        K ← sin(z)    L ← y_r
RPDEC                ROM Pointer = 2H
SPCBI1 ;             RAM1 uses Base1 + Index1
                     (RAM1 = 11H)
-----------------------------------------------------------
SUBF WR2, M          WR2 = −sin(z) * y_r
MOV KLR1, ROM        K ← cos(z)   L ← y_i
SPCBP0 ;             RAM0 uses Base0 (RAM0 =
                     1H)
-----------------------------------------------------------
ADDF WR2, M          WR2 = −sin(z) * y_r + cos(z) *
                     y_i
MOV RAM0, WR0        RAM0(1H) ← WR0 (output
                     real part of X)
SPCBP1               RAM1 uses Base1 (RAM1 =
                     1H)
SPCBI0 ;             RAM0 uses Base0 + Index 0
                     (RAM0 = 11H)
-----------------------------------------------------------
ADDF WR2, RAM1       WR2 = x_i − sin(z) * y_r +
                     cos(z) * y_i
MOV RAM0, WR1        RAM0(11H) ← WR1 (output
                     real part of Y)
SPCBP0 ;             RAM0 uses Base0 (RAM0 =
                     1H)
-----------------------------------------------------------
SUBF WR3, IB         WR3 = x_i − (−sin(z) * y_r +
                     cos(z) * y_i)
MOV NON, WR2         Use value on internal data
                     bus (IB)
INCBRP ;             Increment ROM Pointer by 2
                     (uses special 2^n)
-----------------------------------------------------------
MOV RAM1, WR2        RAM1(1H) ← WR2 (output
                     imaginary part of X)
SPCBI1 ;             RAM1 Uses Base1 + Index1
                     (RAM1 = 11H)
-----------------------------------------------------------
MOV RAM1, WR3        RAM(11H) ← WR3 (output
                     imaginary part of Y)
SPCBI0               RAM0 uses Base0 + Index0
                     (RAM0 = 11H)
INCBP0               Increment Base Pointer 0
                     (BP0 = 2H)
INCBP1 ;             Increment Base Pointer 1
                     (BP1 = 2H)
```

Figure 9. Code for the FFT decimation-in-time butterfly.

tion in time, the computed values of x and y can be stored back in their original locations. By simply modifying the base and index pointers, the algorithm can compute another butterfly with the same instructions it used to compute the first butterfly. Figure 9 shows the code for the FFT butterfly.

In an FFT on the ASP, the real part of a value is stored in RAM0 while the imaginary part is stored in RAM1. The "twiddle factors" are known constants and therefore are stored in the internal coefficient ROM. The data are represented as follows:

Input data:

$x = x_r + jx_i$ and $y = y_r + jy_i$
(r = real, i = imaginary)

Twiddle factor:

$\mathrm{W}k = \cos(z) - j\sin(z)$
(where z is a function of the order of the FFT performed)

Output data:

$X = X_r + jX_i$ and $Y = Y_r + jY_i$

where

$$X_r = x_r + \cos(z) * y_r + \sin(z) * y_i$$
$$X_i = x_i + \cos(z) * y_i - \sin(z) * y_r$$
$$Y_r = x_r - \cos(z) * y_r - \sin(z) * y_i$$
$$Y_i = x_i - \cos(z) * y_i + \sin(z) * y_r$$

In our example the butterfly takes 12 instruction cycles. In general, an n-point FFT will require $n/2$ butterflies per stage and $\log_2(n)$ stages. At 150 ns per instruction, a 512-point complex FFT should take

$$(512/2) \text{ butterflies/stage} * \log_2(512) \text{ stages} * 12 \text{ cycles} * 150 \text{ ns}$$
$$= 256 * 9 * 12 * 150 \text{ ns}$$
$$= 4.1 \text{ ms.}$$

The actual benchmark is 4.7 ms, including the input and output of the 512 complex values and the modification of the base and index pointers.

Availability

The μPD77230 ASP is supported by several development tools. A relocatable assembler is currently available and runs under MS-DOS and CP/M; it will also be available on the VAX/VMS and Unix operating systems by the second quarter of 1987. A comprehensive software simulator is currently available on the VAX. In addition, NEC provides an in-circuit emulator, the EVAKIT 77230, that performs real-time debugging on a target system. The μPD77230 device itself is in mass production now.

Acknowledgments

I thank Mark Davis for reviewing the draft versions of this article and for the timely suggestions, input, and support he provided.

References

1. B. Eichen, M. Davis, and B. Kulp, "Floating Point Math Integrated on Chip Makes DSP IC a Standout," *Electronic Design*, Feb. 20, 1986.
2. L. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic Publishers, Boston, 1986.
3. *μPD77230 User's Manual*, NEC Electronics, Mountain View, Calif., Feb. 1986.

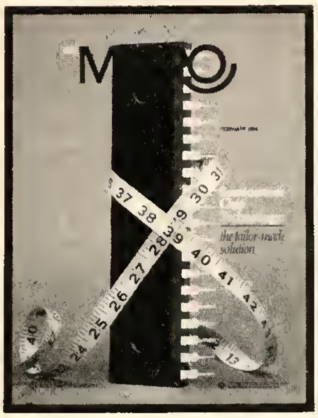
Bill Eichen is a technical marketing engineer in the digital signal processing group at NEC Electronics. Before joining NEC, he was a design engineer in Motorola's corporate research center for speech products. Eichen received a BSEE and MSEE from MIT.

Questions about this article can be directed to Mark Davis, Technical Marketing Manager—DSP, NEC Electronics, Digital Signal Processing Group, PO Box 7241, Mountain View, CA 94039.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 159 **Medium** 160 **Low** 161



IEEE Micro Annual Index Volume 6, 1986

This index covers all technical items that appeared in this periodical during 1986, and items from prior years that were commented upon or corrected in 1986. The index is divided into an Author Index and a Subject Index, both arranged alphabetically.

The *Author Index* contains the primary entry for each item; this entry is listed under the name of the first author and includes coauthor names, title, location of the item, and notice of corrections and comments if any. Cross-references are given from each coauthor name to the name of the corresponding first author. The location of the item is specified by the journal name (abbreviated), year, month, and inclusive pages.

The *Subject Index* contains several entries for each item, each consisting of a subject heading, modifying phrase(s), first author's name, and enough information to locate the item. For coauthors, title, comments, and corrections if any, etc., it is necessary to refer to the primary entry in the Author Index.

AUTHOR INDEX

A

Allison, Paul, *see* Curtis, T. W., *M-M Jun 86* 61–71

Appelbaum, Joseph, *see* Gabbay, David, *M-M Feb 86* 16–23

Aylor, James H., Barry W. Johnson, and Bruce J. Rector. Structured design for testability in semicustom VLSI; *M-M Feb 86* 51–58

B

Baker, Marc A., and Vincent J. Coli. The PAL20RA10 story—The customization of a standard product; *M-M Oct 86* 45–60

Bell, Wayne D., *see* Cooper, Thayne C., *M-M Aug 86* 53-58

Berglund, Eric J. An introduction to the V-system; *M-M Aug 86* 35–52

Bradley, Jon, *see* Frantz, Gene A., *M-M Dec 86* 10-28

Busigin, Anthony. Comments on 'A signal processing implementation for an IBM PC-based workstation' by N. K. Riedel, et al.; *M-M Feb 86* 6 (Original paper, Oct 85)

Butler, James M., and A. Yavuz Oruc. A facility for simulating multiprocessors; *M-M Oct 86* 32–44

C

Cinotti, Tullio Salmon, *see* Neri, Giovanni, *M-M Feb 86* 7–15

Cohen, Brad, and Ralph McGarity. The design and implementation of the MC68851 paged memory management unit; *M-M Apr 86* 13–28

Coli, Vincent J., *see* Baker, Marc A., *M-M Oct 86* 45–60

Colley, Stephen, *see* Hayes, John P., *M-M Oct 86* 6–17

Cooper, Thayne C.. Wayne D. Bell, Frank C. Lin, and Norm J. Rasmussen. A benchmark comparison of 32-bit microprocessors; *M-M Aug 86* 53-58

Corsini, Paolo, and Cosimo Antonio Prete. Multibug: Interactive debugging in distributed systems; *M-M Jun 86* 26–33

Curtis, T. W., Paul Allison, and James A. Howard. A Cordic processor for laser trimming; *M-M Jun 86* 61–71

D

Dirvin, Rhonda Alexis, and Arthur R. Miller. The MC68824 token bus controller: VLSI for the factory LAN; *M-M Jun 86* 15–25

Dubois, Michel, *see* Gaudiot, Jean-Luc, *M-M Oct 86* 18–31

E

Eichen, Bill. NEC's µPD77230 advanced digital signal processor; *M-M Dec 86* 60-69

Epstein, Lev, *see* Gavrielov, Moshe, *M-M Apr 86* 6–12

F

Foer, Debra K., *see* Kahaner, David K., *M-M Jun 86* 52–60

Frantz, Gene A., Kun-Shan Lin, Jay B. Reimer, and Jon Bradley. TMS320C25: Texas Instruments' CMOS 10-MIPS digital signal microcomputer; *M-M Dec 86* 10-28

Fuccio, Michael L., and Benjamin Ng. Hardware architecture considerations in the WE32100 chip set; *M-M Apr 86* 29–46

G

Gabbay, David, and Joseph Appelbaum. The design of a dc/ac inverter with the MC68HC11 microcomputer; *M-M Feb 86* 16–23

Gaudiot, Jean-Luc, Michel Dubois, Liang-Teh Lee, and Nadim G. Tohme. The TX16: A highly programmable multimicroprocessor architecture; *M-M Oct 86* 18-31

Gavrielov, Moshe, and Lev Epstein. The NS32081 floating-point unit; *M-M Apr 86* 6–12

Grossner, Clifford P., Thiruvengadam Radhakrishnan, and Alex Schena. An intelligent braille display device; *M-M Jun 86* 43–51

H

Hayes, John P., Trevor Mudge, Quentin F. Stout, Stephen Colley, and John Palmer. A microprocessor-based hypercube supercomputer; *M-M Oct 86* 6-17

Hearn, Robert J., *see* Jackson, William A., *M-M Aug 86* 18–34

Horlick, Jeffrey, *see* Kahaner, David K., *M-M Jun 86* 52–60

Howard, James A., *see* Curtis, T. W., *M-M Jun 86* 61–71

Huang, Victor K. L., ***Guest Ed.***, and Priscilla M. Lu, *Guest Ed.*. Operating systems (Special issue intro.); *M-M Aug 86* 6–7

J

Jackson, William A., Paul C. Rogers, Robert J. Hearn, and Jeffrey S. Mattiace. Performance and availability in a network file server; *M-M Aug 86* 18–34

Johnson, Barry W., *see* Aylor, James H., *M-M Feb 86* 51–58

Johnson, Barry W., ***Guest Ed.***. Multiprocessing (special issue intro.); *M-M Oct 86* 5

Joshi, Sunil P. High-performance networks: A focus on the fiber distributed data interface (FDDI) standard; *M-M Jun 86* 8–14

+ Check author entry for coauthors

† Check author entry for subsequent corrections/comments

SUBJECT INDEX

+ Check author entry for coauthors

† Check author entry for subsequent corrections/comments

MicroLaw

Richard H. Stern/Law Offices of Richard H. Stern, 2101 L Street NW, Suite 800/Washington, DC 20037

Software copyright developments

Structure of computer programs protected by copyright; implications for "clean rooms"; microcode protectable by copyright

As reported in the last issue of *IEEE Micro* (p. 66), the US appellate court in Philadelphia held that copyright protection of a computer program goes beyond just the lines of code contained in the computer program. It also covers some aspects of the "structure" of the program.

In *Whelan Associates, Inc. v. Jaslow Dental Laboratory, Inc.*, the trial court found the defendant guilty of copyright infringement because the visual screens displayed by the two parties' programs were almost identical in format, and naive users and prospective customers could not tell the difference between what the two systems did.[1] The court of appeals has now upheld this judgment, but on the basis of somewhat different reasoning. The bottom line, however, is that the plaintiff/copyright owner in this case is now adjudged to be entitled to 75 years of exclusive right to market computer programs having the structure (however that term is defined) of the registered program.

There was no copying of code in this case. Among other things, the computer programs were written in different languages, so that there could not be any byte-for-byte or other literal copy of the code. The original program was written in EDL, or Event Driven Language, and the infringing program in Basic. It is unclear from the opinions whether it is possible to "translate" an EDL program into Basic, or whether instead the program must be greatly rewritten to port it. In any event, it appears to have been conceded or assumed here that the code of the infringing program was not a 1:1 mathematical transform of the original code.

The court of appeals did not explain what it meant by a computer program's structure in general or what aspects of program structure are to be protected against a second comer. In its opinion the court did give examples, however, of objectional copying of structure. A prominent example was the "file structure," or what we might term the selection of *fields* (for database types, *attributes* for more theoretical computer science types). In the case of what seems to be an invoicing program, the defendant appears shamelessly to have copied from the plaintiff the plaintiff's original selection of all of the following fields:

- description of item sold,
- number of items sold (n),
- unit price (p)
- extension ($e = n * p$), and
- total amount to be billed (sum of e's).

In addition the defendant imitated plaintiff's practice of setting a flag in the record after invoicing a customer, to prevent reinvoicing the customer for the same bill.

In these blatant circumstances the court of appeals felt that it had no alternative but to hold the defendant guilty of copyright infringement for copying the plaintiff's creative work. Although the court did not mention it, probably the copying of format went even farther and extended to such plagiarism of structure as putting the customer's name and address on the invoice and the date on which the invoice was prepared.

What the court did, as this example shows, should give considerable pause to would-be copyists of commercially successful computer programs. What the court said by way of explanation should slow them down even more.

Before the court could hold the file structures of the plaintiff's computer program to be protected by the copyright registration here, the court had to assure itself that it was protecting the "expression" rather than "idea" of the program. This is a fundamental—perhaps *the* fundamental—principle of copyright law: Copyrights protect authors' expressions of ideas, not their ideas themselves.

Therefore, despite the absence of any examination of a work for technical advance or creative merit (as occurs under the patent system), there should be no public concern over proliferation of monopoly because of grants of copyright protection. Such monopolies cannot occur, because everybody else is free to create and secure a copyright in his own individual expression of the same idea, and an infinity of different possible expressions of any idea exists.[2] Ideas remain freely available to everyone, and they are not locked up by the copyright laws; only authors' particular personal expressions of ideas are reserved by copyrights.

The court conceded that "it is frequently difficult to distinguish the idea from the expression" in a work of authorship, or even "elusive." But the court found that it was easy to do so in the case of a computer program, by means of a new legal test that the court had devised. Any choice in writing the program that is not necessary to achieve the purpose of the program is expression; any choice necessary to achieve the purpose is idea.

This "rule," of course, simply changes the discussion from what is idea and what is expression to what is necessary and what is not necessary to achieve the purpose of the program, so that the definition of the two terms, "necessary" and "purpose," must be addressed. For some reason the court felt that thus shifting the focus of discussion would further the analysis.

Of the two terms, purpose is by far the less objectively definable; it is indeed the pea under the shell at the legal carnival. Purpose, like idea, can be defined narrowly or broadly; it is an accordionlike concept. It can be a precise species or the most indefinite genus.

For example, what is this yellow-colored object I hold in my hand? Is it an instance of tangible matter, organic matter, vegetable matter, a fruit, a citrus fruit, a lemon, or what? Suppose that it matters, in the context of a court's decision whether Peter must pay money to Paul, whether the object that you hold in your hand is the "same" object as mine. Must Peter pay if you have a goldfish, a squash, a banana, an apple, a grapefruit, or another lemon? And if it is another lemon, what if one is a California lemon and the other a Florida lemon, a lemon from the lemon tree adjacent to mine, or from the next branch on the same tree? And if I hand you my lemon, is it still the same lemon a few minutes or days later? Suppose, further, that you enhance, adapt, debug, or otherwise modify the lemon?

Now I suppose that you can say that there are practical answers to these metaphysical questions, so that whether you hold a fish, a squash, a banana, or a debugged and enhanced lemon, the answer to the question of what our held objects are (or to what category we will assign them in our discussion) will depend on the context in which the question is asked. Are we concerned with paperweights, missiles, food, dessert, main course, something to put into iced tea, or what?

But that is *not* how the court of appeals approached the question of defining the purpose of these programs. The court said that the application of its newly proposed rule of law presented no difficulties in the present case, because the true purpose of the parties' computer programs was so clear. The purpose was to "aid in the business operations of a dental laboratory," or to put it with even greater precision "simply to run a dental laboratory in an efficient way." The structure of the computer program "was not essential to that task." Accordingly, the structure was expression rather than idea, and the defendant thus committed copyright infringement by copying it as described above.

It requires little reflection to conclude that when the purpose of a computer program is defined that generically few or no features of the structure of computer programs will ever be necessary to achieve that purpose. Indeed, the purpose test becomes a sham or charade, for it inevitably leads to the same result: guilty as charged. The purpose test is not a tool for analysis; it is a ritual incantation that could just as well be omitted for all the difference it makes in the outcome. When purpose is stated broadly enough, nothing is then essential to achieve the purpose.

Thus, under the *Whelan* court's concept of purpose, not even mailing an itemized invoice is "essential" to running a dental laboratory. Each item could have been sold COD or by a cash or credit-card sale. Or a messenger could be sent to the customer to make offers he could not refuse, such as, pay for the merchandise right now or have your kneecap broken. That is certainly one concept of efficiency.

Even if the purpose of the computer program had been defined so that it was not unacceptably broad, that would not have made the test of being necessary or essential to achieve the purpose a correct test. For example, suppose we instead consider the purpose of the invoicing program to be to make it possible for naive dental-lab users to use a microcomputer in preparing invoices without having to understand much about computers or software. Something could both be unnecessary to achieve that purpose and be an idea that the copyright laws do not (and should not) permit anyone to own. Possible examples are writing the program in C to make it run fast, using friendly icons, having the program driven by a menu in which the user moves the cursor to a choice and presses <Enter> instead of typing in an alphanumeric code shown on the screen, and using the address on the invoice to prepare a mailing label with the same address.

There is simply no logical implication that because something is not necessary other people should refrain from using it lest they be liable for copyright infringement. To be sure, probably nothing that is necessary should be protected by a copyright, as contrasted with a patent, but the converse is not true. The public domain includes many "unnecessary" things. Under a free-enterprise system the government does not prohibit businessmen from doing particular things just because someone else did them first; such a government prohibits business from free exercise of volition only when there is a stated reason for the compulsion, usually the realization of some recognized public purpose.

The analysis in the *Whelan* opinion is so bad that it is difficult to know where to start criticizing it. Indeed, the court seems to have only one thing right, which I have not mentioned so far but should. The court was concerned with evidence that coding in general is, and in the case of these particular programs was, responsible for only a small fraction of the total time and cost of developing the computer program—perhaps 20 percent. Organizing the dataflow, partitioning out repetitive subroutines, and other noncode aspects of the programs accounted for "a tremendous amount of time" in developing the plaintiff's computer programs. The court indicated that the noncode aspects of the computer programs may have embodied most of the creativity and commercial value of the programs. Clearly, this view of the facts caused the court to conclude that these noncode aspects should be legally protected and that the defendant should be made liable to the plaintiff for appropriating them.

The factual premise is doubtless defensible, perhaps on balance the better view when properly refined, but the conclusion of copyright infringement is a *non sequitur* and the court's legal analysis is still bad. That program structure (whatever that term means) is commercially valuable, maybe even far more valuable than the particular coding, does not mean that the copyright statute was intended to or should cover that structure. Perhaps something should protect structure, but probably *not* something just like the copyright laws, and certainly not something that protects the kind of structure on which the court fastened its attention.

Any system that protects things like the field selection of the invoicing program described above is a terrible idea. Use of those fields is either already part of the public domain or a trivial variation on it, and everybody should be free to use it commercially; no one should have the right to prevent competitors from "appropriating" that kind of structure. Perhaps, field selection in a database system should never be protectable.

Other noncode aspects of software, however, may well be ideas that deserve some sort of protection. Possible examples are algorithms, flowcharts, instruction sets, languages, and perhaps metaphors and icons. But their "misappropriation" does not call for 75 years of injunctions, criminal penalties, and all the rest of the copyright arsenal. A more modest proposal would be in order.[3]

The *Whelan* court extrapolated from a probable social-economic need to a solution of its own devising. Courts are not suited for that task, and it is probably beyond the scope of the role assigned to courts under our system of government. In this case, moreover, the court's solution was poorly conceived and likely counterproductive as well.

The *Whelan* decision may have serious implications for the "clean-room" theory of legally writing a BIOS or other software. The clean room is an expensive procedure for reinventing the wheel. Under this theory shadow or form is exalted over substance, because it is considered that the applicable copyright law is all shadow and no substance. The clean-room procedure for writing competitive software (e.g., an IBM PC clone's BIOS, a dBase substitute, a Lotus 1-2-3 clone) works as follows:

The Whelan *decision may have serious implications for the "clean-room" theory of legally writing a BIOS or other software.*

- Teams A and B are set up to develop an emulator of the target software. Team A consists of ordinary systems analysts. Team B consists of computer programmers who have never seen the code of the target software. (It is said that they must come via spaceship directly from a distant planet.)
- Team A begins to analyze the target software. Team B is locked up in the clean room and protected from contamination by contact with Team A or the target software.
- Team A studies the target software in detail. It determines the specifications. It may disassemble the code, reverse engineer the flowchart, and otherwise learn how the computer program works.
- Team A writes a report describing the specifications and requirements of the computer program but providing no code. The report is supposed to be all idea and no expression.
- The report is passed to Team B via a porthole into the locked clean room. Team B then writes the code called for in the report. Only then is Team B allowed to leave the clean room and become contaminated.
- Team B never had access to the actual code, either directly or by being told what it was by Team A. Team B has had access only to the ideas of the target software. The conclusion sought to be drawn is that any parallelism in code is either

Drawing the line

The following comments are offered by Michael A. Dailey and Henry W. Jones III, two Atlanta attorneys associated with Microstuf Inc. in its suit against SoftKlone Distributing Corp. over the latter's Mirror clone of Microstuf's Crosstalk communications program. The suit is now pending in the federal district court in Atlanta. (*Ed. note: See "MicroReview" in this issue for Dave Hannum's review of Mirror.*)

"Clone makers" have tried to seize upon the recognized principle of US copyright law that blank forms are not proper subjects for copyright protection. The clone makers have tried to exploit this principle as a justification for copying the formats of screen displays and other user interfaces on which software innovators have lavished vast sums of money and hours of toil. But the clone makers disregard the notable exception to this rule, which holds that forms that not only record information but also convey information are worthy of copyright protection.

As applied, this rule has protected such things as hospital examination forms, legal forms, forms for gasoline station account books, and answer sheets for psychological tests. These precedents are instructive in showing that computer screen formats may also be protected by copyright. And just as the courts have protected the "look and feel" of greeting cards, video games, and other pictorial or audiovisual works, we believe that so too should they protect the look and feel of user interfaces.

That theory is being tested in Microstuf's pending case against SoftKlone over the Crosstalk and Mirror communications programs. Mirror is an emulator of Crosstalk XVI, version 3.6. A comparison of the two programs' screens (see Figure 1) reveals identically expressed communications parameters, as well as Filter, Key, and Send control settings. Mirror also duplicates all 87 commands used in Crosstalk and the biliteral alphanumerics used to abbreviate them for user keyboard entry.

We feel that this case, and other cases like it pending in other courts around the country, will determine whether software clone makers will be free to copy the creative work of software originators in devising screens and other user interfaces. We consider the *Whelan* case a significant pointer toward increased judicial recognition of the importance of protecting the creativity of originators and in requiring clone makers to adjust their product development strategies accordingly. For example, we believe that the *Whelan* decision indicates that for Mirror to mimic program features of Crosstalk that are not essential to the proper functioning of a communications program is a copyright infringement, and therefore it should be enjoined by the courts and made subject to damages.

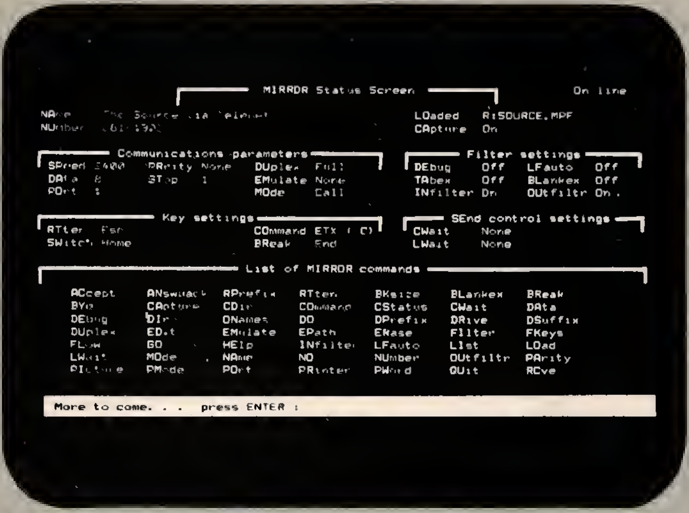
This will not impair competition in the delivery of software to the public, because there are many alternative means of expressing the necessary commands and other features of such programs, and the public can just as readily learn one as the other. For a second comer to take a free ride on the time, effort, and expense of a software innovator in educating the public to the usefulness of a new type of program and in how to use its user interface is simply misappropriation of the business values of the innovator. That kind of reaping where another has sown will and should be stopped by the courts, as the *Whelan* court recognized and as its decision provided.

M.A.D. and H.W.J. III

Reader comment on these views is welcome. Do *IEEE Micro*'s readers feel that it is more important financially to encourage and thus to stimulate software innovation or that it is more important to encourage price competition by clone makers? Who "owns" and is entitled to profit from the users' efforts in having learned a command set or interface? Is it the creator of the command set or interface, the user, or anybody who wants to come along and exploit the user investment in learning how to use the interface?

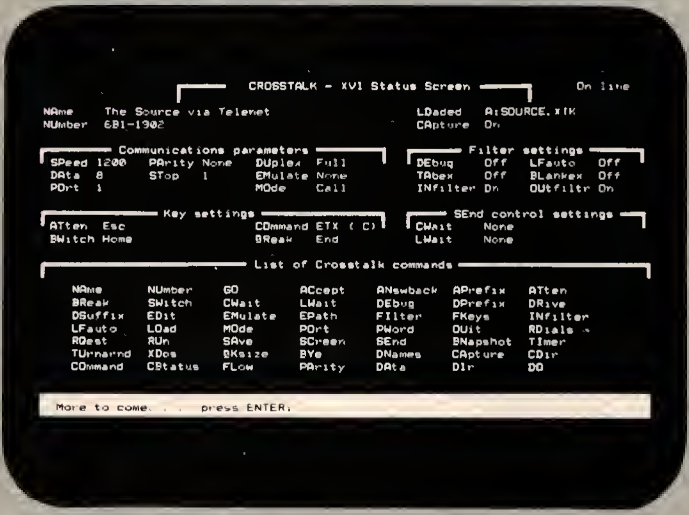
Should anyone who tries to use F1 for anything but calling up the help menu be suppressed? Should anyone who claims a monopoly on the use of F1 for help be suppressed? Should we all agree to use F1 for help, F9 for recalculate, Q for quit, Y/N for yes or no?, and so on, and do so freely? Or should the first user be the only user, or at least be compensated by all later users?

In the case of the biliteral terms seen in Figure 1, should the second comer be forced to devise his own two-letter combinations for BReak, DEbug, DIr, EDit, NUmber,



```
                  MIRROR Status Screen                     On line
NAme     The Source via Telenet               LOaded   A:SOURCE.MPF
NUmber   681-1902                             CApture  On

 Communications parameters                     Filter settings
 SPeed 2400   PArity None   DUplex  Full       DEbug     Off   LFauto   Off
 DAta  8      STop   1      EMulate None       TAbex     Off   BLankex  Off
 POrt  1                    MOde    Call       INfilter  On    OUtfiltr On

 Key settings                                  SEnd control settings
 ATten  Esc                 COmmand ETX ( C)   CWait     None
 SWitch Home                BReak   End        LWait     None

                        List of MIRROR commands
   ACcept    ANswback  APrefix   ATten     BKsize    BLankex   BReak
   BYe       CApture   CDir      COmmand   CStatus   CWait     DAta
   DEbug     DIr       DNames    DO        DPrefix   DRive     DSuffix
   DUplex    EDit      EMulate   EPath     ERase     FIlter    FKeys
   FLow      GO        HElp      INfilter  LFauto    LIst      LOad
   LWait     MOde      NAme      NO        NUmber    OUtfiltr  PArity
   PIcture   PMode     POrt      PRinter   PWord     QUit      RCve

 More to come. . .   press ENTER :
```

(a)



```
                  CROSSTALK - XVI Status Screen            On line
NAme     The Source via Telenet               LOaded   A:SOURCE.XTK
NUmber   681-1902                             CApture  On

 Communications parameters                     Filter settings
 SPeed 1200   PArity None   DUplex  Full       DEbug     Off   LFauto   Off
 DAta  8      STop   1      EMulate None       TAbex     Off   BLankex  Off
 POrt  1                    MOde    Call       INfilter  On    OUtfiltr On

 Key settings                                  SEnd control settings
 ATten  Esc                 COmmand ETX ( C)   CWait     None
 SWitch Home                BReak   End        LWait     None

                        List of Crosstalk commands
   NAme      NUmber    GO        ACcept    ANswback  APrefix   ATten
   BReak     SWitch    CWait     LWait     DEbug     DPrefix   DRive
   DSuffix   EDit      EMulate   EPath     FIlter    FKeys     INfilter
   LFauto    LOad      MOde      POrt      PWord     QUit      RDials
   RQest     RUn       SAve      SCreen    SEnd      SNapshot  TImer
   TUrnarnd  XDos      BKsize    BYe       DNames    CApture   CDir
   COmmand   CStatus   FLow      PArity    DAta      DIr       DO

 More to come. . .   press ENTER:
```

(b)

Figure 1. Status screens of (a) Mirror and (b) Crosstalk XVI.

PArity, QUit? Should he come up with new names for those commands, so that other biliteral terms will seem appropriate? Are there practical alternatives for the examples given? If there are, would it really be a good idea to introduce them?

What about more subtle things, such as whoever thought of using labels (e.g., *:label*) instead of absolute line numbers to send a GOTO to the right line for a subroutine or branch? Should we all pay him or her, or just let him or her bask, unpaid, in the satisfaction of having benefitted humanity?

How far should this reasoning go? Can we draw any line sensibly between acceptable and unacceptable copying by emulators/clone makers?

```
Can the legal system deal sensibly with this sort of thing?      :How
  If yes, by copyright?                                          How?
    If yes, GOTO :Buy                                            Write if you get work
    If no, GOTO :How                                             Good luck
  If no, can something else?                                     GOTO :End
    If yes, GOTO :What
    If no, GOTO :End                                             :What
                                                                 What? And
:Buy                                                             GOTO :How
Buy gold brick, Brooklyn Bridge, etc., from nice man
GOTO :End                                                        :End
```

purely coincidental or the effect on the code of the ideas passed through the porthole (Team A's report).

Whether or not the clean-room procedure is a silly charade, many lawyers and MBAs swore by it. It was the magic formula for avoiding liability for copyright infringement. But under the *Whelan* rule, what can Team A now pass to Team B via the porthole of the clean room? It would now seem that anything of value that is passed will be expression, since idea now includes only the most general and abstract formulation of the purpose of the computer program; the "unnecessary" rest of the description of the target computer program and the least detail beyond the abstract purpose will be expression. Well, just as one good turn deserves another, perhaps one silly concept deserves its comeuppance from another.

A federal trial court in San Jose has held that microcode is copyrightable. NEC had sued Intel for a judgment declaring that the 8086/8088 microcode either was uncopyrightable or that NEC did not infringe it by marketing the V20 microprocessor chip. Intel then countersued for copyright infringement.[4]

The court held that the microprograms stored in the microcode ROMs of the 8086 and 8088 were computer programs under the Copyright Act, and literary works. It said that writing microcode was a "creative endeavor" and that the programming methodology was indistinguishable from that employed in creating other types of computer programs. It also ruled that the fact that microcode has a function or utilitarian purpose does not make microcode uncopyrightable.

The court left for a future date its determination whether the particular NEC microcode in fact infringed the copyright in the Intel microcode. NEC applied to the court and was given permission "to present evidence of 'clean-room' creation of [its] microcode." (Because the San Jose court is not under the authority of the *Whelan* court of appeals, it is not obliged to follow the implications or even the holding of that decision.)

It is unclear if the court has left open the question of whether the NEC microcode is noninfringing because (according to NEC) it copies only "functional" parts of the Intel microcode. The court held that "the function performed by

[Intel's] 8086/8088 microprograms does not affect their status as copyrightable subject matter.'' But whether the copyright in utilitarian works extends to and protects their functional aspects is a separate question from whether the existence of such aspects makes the works uncopyrightable. There have been cases in which the courts held that it was not copyright infringement to copy the functional aspects of utilitarian works, while in other cases courts have held that utilitarian works were uncopyrightable because of their functionality.

Menus were protected in a recent ''look-and-feel'' copyright infringement decision from the San Francisco federal trial court. Broderbund Software sued Unison World (now Kyocera Unison) for copying the screens of Broderbund's Print Shop display graphics program. Unison's Print Master was the product of a fallen-apart joint venture in porting Print Shop from the Apple to the IBM PC.

According to the court, the infringing screens tracked the appearance of the originals in many arbitrary, nonfunctional aspects, such as relative size of characters on the display, layout, and choice of wording for phrases. The court concluded that an ''ordinary observer could hardly avoid being struck by the eerie resemblance between the screens of the two programs,'' since ''the sequence of screens and the choices presented, the layout of the screens, and the method of feedback to the user are all substantially similar.'' The court gave a list of parallels, some of which seem to show nothing (for example, that both products require the user to create the front of a greeting card before the second page) and others of which seem to show copying of aribitrary details (for example, division of a page into 13 sectors).

The case differs from other look-and-feel cases in that the basis for the court's judgment of copyright infringement was not a ''literary'' or textual copyright in the code, but a pictorial type of copyright in the screens themselves. The screens were registered as audiovisual works, so that the look and feel in controversy was that of the pictures as pictures rather than that of the computer program as such. That is an important difference. It is generally recognized that pictures may have a difficult-to-articulate but nonetheless protectible look and feel. It is more open to question, however, that a wholly symbolic work has a protectible look and feel that is anything other than its unprotectible concept or its ''ideas.''

At first blush the copying here seems excessive and unnecessary. It does not seem to be dictated by a need to conform to user habit or by some other functional consideration. However, some industry members claim that the decision will adversely affect the development of enhanced competitive products.

In summary, in the past few months there has been a considerable expansion of the scope given copyrights in protecting various noncode aspects of computer programs.[5] In some instances, such as the copying of screens copyrighted for their pictorial content, the development seems incremental and unobjectionable. In other cases, however, the result seems to be that patentlike protection of functionally important or public-domain subject matter is being awarded, merely on the basis of a copyright registration without any examination by an expert body, or even a court, into whether the work displays enough technical advance or merit to justify the award. The result may be to deprive other software developers of the privilege of using things (1) that contribute to making their own technological advances, to the public benefit, and (2) in which they may be properly entitled to share, unless and until the first comer is able to obtain a patent on the feature.

Courts protecting look and feel have a need to protect commercial values of software innovators.

The courts protecting look and feel or other noncode aspects of computer programs have responded to a need that they perceived to protect commercial values of software innovators. They have been convinced, probably rightly, that a major portion of the time, effort, and expense that goes into developing a computer program concerns noncode aspects of the program. These aspects may include user interfaces, menu choices, program metaphors (and icons, such as the famous or infamous file-deletion garbage can), and other things termed ''structure.'' The courts have also been persuaded that protecting these things under the copyright law will promote technological progress in software, to the benefit of the public, and that refusing to allow such protection will discourage investment in software innovation. The empirical evidence for this has been slender, and it is unclear whether the net effect of applying copyright law in this manner will be a plus or a minus on software progress.

Applying copyright law is the only available quick patch, for no other convenient federal system exists right now.[6] Many proponents of such protection are unwilling to await legislative action, for it may never come or they may go bankrupt as a result of competition from ''software clone makers'' in the meantime. Clone makers, of course, take a different view; they say that this patch is a kludge. So, too, may more disinterested observers. The following is a quotation from a recent essay by Congressman Robert Kastenmeier, the chairman of the House subcommittee responsible for intellectual property matters:

> In studying the problem of how best to protect semiconductor chip products and in devising a legislative scheme that solves the problem and also promotes the public interest, I found the study and the solution to be a paradigm of the industrial property protection problem for all new technology at the end of the Twentieth Century and at the rise of what may be a new information society. One of the things that I believe we learned was that the era of ''shoehorning,'' or of pouring new wine into old legislative bottles, should end. We learned...that different bodies of intellectual property law strike different respective equilibrium points for balancing the interests and values at stake, that what is an acceptable or desirable balance of interests for authors and artists is not necessarily acceptable in the case of industrial property products, and that ''it would be pure serendipity for a law designed to deal with literary and artistic rights to realize the needs of new technology," We are no longer so blindly self-confident as to expect that such a serendipitious result will automatically occur....[7]

References

1. For a discussion of the trial court's opinion, see *IEEE Micro*, Apr. 1985, pp. 88-89.

2. For a discussion of whether there are many or few ways to express certain computer programming ideas, see *IEEE Micro*, Apr. 1984, pp. 69-70 (optimal execution-speed compilation of TMS320 object code from Fortran source code).

3. Probably, more arguments can be made in favor of some degree of protection for noncode aspects of computer programs than can be made against it. But the protection should be more like that for chip layouts under the Semiconductor Chip Protection Act than for books under the Copyright Act. For a more detailed discussion of the problem, and a proposal for protection of instruction sets, algorithms, icons, and other noncode "idea" aspects of software, see the author's paper delivered at the Software Engineering Institute 1986 Spring Symposium, Pittsburgh. It can be found in more fully developed form in the current issue of the *University of Pittsburgh Law Review*, "The Bundle of Rights Suited to New Technology," Vol. 47, p. 1229, 1986.

4. For a discussion of the issues in this case, see *IEEE Micro*, Apr. 1985, pp. 89-92.

5. See accompanying box and *IEEE Micro*, "MicroLaw," Apr. 1986, pp. 64-65, for discussions of other pending look-and-feel cases.

6. Some states have "unfair competition" or "misappropriation" laws that might be interpreted to protect noncode aspects of software. But there are several problems with this approach. First, not all state laws are amenable. There is a nonuniformity problem for multistate marketing.

Another problem is possible "preemption."' The federal copyright law sets aside state laws that are equivalent to copyright law or offer similar protection for subject matter within the general scope of copyright law. Whether or not copyright law will ultimately be held to cover noncode aspects of software, Congress probably has the power to make copyright law apply to them if it so desires. Hence, duplicative or inconsistent state protection of the same general subject matter may well be preempted.

7. R. W. Kastenmeier, "Foreword to R. Stern's *Semiconductor Chip Protection*, Law & Business/Harcourt Brace Jovanovich, 1986, p. xxv.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 174 **Medium** 175 **Low** 176

Department

MicroStandards

Michael Smolin/Smolin & Associates/3428 Greer Road, Palo Alto, CA 94303

Publish and/or Perish (Or, Who Wants To Use a Trial-Use Standard?)

In the annals of the Computer Society lie numerous stories of the attempts to publish drafts of proposed standards. Few of these attempts were successful—most were (and are) not. The arguments, pro and con, about publishing draft standards have taken on religious overtones. Adherents to each argument may even refuse to see little, or any, value in the other arguments.

Typically, a working group in the process of developing a standard needs to publish a draft to solicit comments from a broad base of peers. Standards administrators in the IEEE and especially in the Computer Society fear that, no matter how disclaimers are worded, someone will insist on mistaking the published draft for an adopted standard.

Let's examine the positions with a hypothetical dialogue...

Standards developer (SD): The most important feature of IEEE standards is that they are standards of *consensus.* We must see to it that "every attempt is made to involve all interests in the activity" so that "it can be presumed that the document represents a consensus of all interests concerned with the scope of the standard."[1]

The working group developing the standard and the sponsoring technical committee often feel that true consensus requires broadly circulating the draft to stimulate comments from all facets of the interested parties. This is done best by printing the draft under consideration in a widely read professional publication such as *IEEE Micro* or *Computer.*

Standards Administrator (SA): There are other ways to get the wide distribution you feel you need—without publishing the draft of a proposed standard. You can publish articles about the draft and about the working group's resolution of opposing interests. Or, you can recommend that the proposed draft be adopted as a trial-use standard.

A trial-use standard has a two-year life span. During that time, it will be treated by the IEEE as a true standard. It will be published and offered for sale by both the IEEE and the Computer Society. Comments received by the working group during the trial-use period represent the public comments that you need to achieve consensus. You then can revise your draft, reballot it, and resubmit it—this time for adoption as a full-use standard.

SD: You mean that by accepting a two-year trial-use period as a delay, we can go for a full standard without really achieving consensus—after all, few in the microcomputer area would bother implementing or examining in detail a "trial-use" document? Its very name implies that it likely will change just about the same time that a new and complying product could get to market. Note also that IEEE standards go on to become ANSI standards and often become international standards.

About an article on a draft—just who will write it? We have already devoted hundreds of our volunteered hours to writing the draft. Now you also want us to write an article about it for publication. My department's budget doesn't stretch that far. I still have to do my regular work and satisfy my managers.

SA: There is great concern that a published draft might be mistaken for an approved standard. This seems to happen even when care is taken to include disclaimers, expiration dates, etc. Official policy about publication of drafts of standards is that "the practice is deprecated by the Standards Board."[2]

SD: There is another reason that we feel publication of drafts of standards should be unhindered—FAIRNESS. Working groups often have few members—perhaps a dozen or so, commonly only a handful of participating members. Now, I mean members of all categories, including nonparticipating observers. These members become an informed elite who have an information advantage over others. This translates into unfair advantage in marketing and technology. If the IEEE and the Computer Society truly serve their professional membership, it should see that all of these standards development activities get widespread dissemination, including the publication of drafts. It's only fair.

SA: Fairness as you see it may be a luxury that we cannot afford. There is no requirement to force the membership to be knowledgeable about standards developments. The publishing of additional hundreds of pages each year in society magazines is a cost not warranted by member interest.

SD: Well, then let us submit the draft for publication in other trade journals after giving the society's publications the first right of refusal. Publishing the draft also updates users about the state of the development of the standard. Many individuals working from third-hand and out-of-date information do not know how the specifications have been changed over several drafts. We do a disservice by not explicitly telling them about the changes.

SA: You can certainly supply them with a current copy of the draft as a working document, upon request. There is no obligation to do that, but each project may (with approval) supply copies of their documents and charge for that service.

Well, working professional, what do you think? Does the publication of draft standards, the resulting spread of information, and the opportunity to contribute comments during development outweigh the risk of mistaking a draft for an approved standard? Or, do you think the risk of accidentally working to an unapproved draft (proposed standard), with the possibility of wasted resources and efforts, outweighs the arguments to publish draft standards?

References

1. *IEEE Standards Manual,* IEEE Standards Office, 345 East 47th Street, New York, NY 10017, 1982, Section 1.1.
2. *IEEE Standards Manual,* Section 11.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 189 Medium 190 Low 191

MicroNews

MicroNews features information of interest to professionals in the microcomputer/microprocessor industry. Send information for inclusion in MicroNews one month before cover date to Managing Editor, IEEE MICRO, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

SDIC conference to explore storage/interface devices, systems architectures, networks

Computer systems designers, integrators, and specifiers, value-added resellers, value-added OEMs, and high-volume end users can look forward to the first meeting next February of the Systems Design and Integration Conference—an event in the planning stage for two years. Specifically designed to focus on practical solutions to the needs of these groups, SDIC will offer three days of technical sessions, tutorials, and exhibits of computer-related products.

Each day of the conference highlights one of three technologies: storage and interface devices, systems architectures, and networks. Phil Devin, senior analyst for Dataquest's Computer Storage Industry Service, will open the first day's session, speaking on that industry, its products, and its market trends. Planned technical sessions include software design and integration for 32-bit microprocessors, architectures for computer graphics (led by *IEEE Micro* editorial board member Richard Mateosian), and storage trends (led by *IEEE Micro* editorial board member Kenneth Majithia).

Walter J. Utz, Jr., Hewlett-Packard software engineering training manager, will open the next day's meeting with comments on the impact of RISC on computer design. Session topics include the impact of GaAs on systems design, 32-plus bus trends and choices, and systems design with 32-bit microprocessors.

SDIC's last day will begin with remarks from J. Edward Snyder, general manager, TRW Information Networks Division. Solving problems with LANs, multivendor systems integration, MAP: the key to an integrated factory, and fiber optics systems are some of the session topics.

Tutorials

SDIC tutorials are designed to help participants learn about the design and implementation of databases, motion control systems, AI/expert systems, and systems design methodologies and CAE/CAD tools, among other subjects. Tutorial instructors include consultant Herb Edelstein of Digital Consulting Associates; Stanford University associate professor Gio Wiederhold; researcher, author, engineer, and consultant Jacob Tal; and the director of University of Santa Clara's Center for Information Storage Technology, Al Hoagland.

Registration information

The Systems Design and Integration Conference is scheduled for February 10-12, 1987, at the Santa Clara Convention Center in San Jose, California. Wescon, the Los Angeles and San Franciso Bay Area councils of the IEEE, and the southern and northern California chapters of the ERA cosponsor the event.

General registration, which includes the opening session and exhibit areas on all three days, is $5. Preregistration fees for opening session, exhibit area, and each day's technical sessions are $80 per day. The seven attendance-limited tutorials cost $150 to $180.

More information on SDIC can be obtained from the conference producers, Electronic Conventions Management, at (213) 772-2965.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 186   Medium 187   Low 188

Editorial Board welcomes new members

Editor-in-Chief James J. Farrell III announced the acceptance of two new members to the *IEEE Micro* editorial board, Marlin H. Mickle and Yoichi Yano. Mickle assumes the duties of New Products editor beginning with this issue.

Marlin H. Mickle is professor of electrical engineering at the University of Pittsburgh, where he has also held the positions of graduate program coordinator and director of the Computer Engineering Program. He is active in the areas of digital computer systems and high-technology applications.

Marlin H. Mickle

Yoichi Yano

Mickle received his BS, MS, and PhD degrees in electrical engineering from the University of Pittsburgh in 1961, 1963, and 1967.

Yoichi Yano has been a microprocessor architecture designer at Microcomputer Products Division, NEC Corporation, since April 1980. During the past four years he participated in the architecture and system designs of the V60/V70 32-bit VLSI microprocessors. His current interests include VLSI processor architecture and highly parallel computing structures.

Letters

continued from page 5

because of the adoption of power-line multiplexing as the predominant method for home device control, which has inadequate bandwidth for computer applications in general. Later the study group effort moved more to a higher performance backplane bus. That was the precursor to Andy's writing the PAR for the 896 project, which he then chaired. Andy preferred to use the more modest nomenclature, "Advanced Backplane Bus," rather than "Future Bus," which the rest of us had used for a long time previously. He is correct in his assertion that the name on the PAR was not Future Bus.

The contribution of Matthew Taub to the arbitration scheme used in Fastbus (IEEE 960), S-100 (IEEE 696), and Futurebus (IEEE P896) was known to us in the 696 working group very early after its suggestion by David Gustavson. Evidently Andy didn't happen to attend the working group meeting at which that information was stated.

The scene depicting the 896 meeting in Boulder was based on information Andy provided to me over the telephone. I'm glad that he points out that Rollie Linser was the inspiration for a serial bus that can substitute for the parallel bus for reliability. Apparently this was not in my notes, and so I tagged the balloon to Andy as chair expressing a general characteristic. That perhaps can be considered artistic verisimilitude—and possibly justifies the question mark in the title after "Historical." Many of the details contained in the presentation were my personal recollections going back as far as a decade and which had no written material anywhere for support. J. D. Nicoud spent a year's sabbatical in Palo Alto in the early eighties and attended MSC meetings while he was here. Andy is correct in saying that he came after the meeting at which Andy resigned.

The statement Andy gives regarding his resignation was because the MSC's "general failure to prevent the holders of minority viewpoints from indefinitely delaying progress." As one of the minority in the 896 working group I'd like to state that we had no intention of indefinitely delaying progress. Rather we felt that the parallel bus as it was then proposed was not a significant advance on the state of the art and had problems with driving the bus lines, problems which were not resolved. I personally was unhappy that Andy saw fit to resign rather than to resolve the differences within the working group.

Now let's see where Andy and I do agree. It is now 1986, eight years after the Futurebus effort started and no 896 draft has been approved by the MSC! I credit Paul Borrill with working extremely hard as chair, as had Andy, but Paul estimated that at most one year would be needed to finish the draft when he assumed the chair. It has taken at least three years more than that. When the realities of industrial competitiveness are taken into account, a company simply may not be able to wait for the IEEE standards development process to take its path, and time. The 802 efforts to develop LAN standards, however, constitutes a good counter example of the computer industry working constructively within the IEEE framework to develop badly needed standards in a reasonable time scale.

What has come out of those three extra years spent on the 896 effort?

- The new bus drivers developed by R. Balakrishnan of National Semiconductor set a new level of performance in backplane buses.
- The Taub arbitration scheme has been enhanced to incorporate the suggestions of Keith Britton improving fairness among competitors.
- A new parallel-bus protocol containing fast two-edged handshakes from Fastbus was worked out by John Theus of Tektronix.
- The parallel-bus protocol was extended to provide services needed by caches.

Are these efforts worth it? Only time will really tell. Some of the other 32-bit buses now in existence have used features first hammered out in the 896 committee. Under Andy, P896 chose the Eurocard format, which VME and MB 2 and Nubus followed. MB 2 has also chosen the Taub arbitration scheme. Maybe in the future some of these other buses will see fit to retrofit to the higher performance possible with the 896 bus drivers.

I did not want to prepare another WOW (Write Only Writing) article, which only the writer would really read and so chose the cartoon format for the presentation. Perhaps Andy is right and professional society journals should not contain such a format. But the content was the best I could recall, and the effort needed to draw it was about three times that which a conventional written presentation would have required.

IEEE standards during the Great Bus Wars—another view

(Editor's note: When I approved the August cartoon-feature MicroStandards column for publication, it was intended to provide the reader with a depiction of the events of the past decade in a light, and I hoped, humorous manner. Any misrepresentation or offense given was not intentional and is regretted. Allison has responded to my request to submit his account of the events he has noted and that account follows.—J.F.)

The August issue of *IEEE Micro* presented one view of the work done within the Computer Society's Microprocessor Standards Committee. As someone who joined that body in November 1977, just three months after its inception, and was an active participant for over four years, I would like to offer a different one.

The genesis of the Microprocessor Standards Committee, originally established as a subcommittee of the Computer Society's Standards Committee, lay in the loss of control by its developer (MITS Inc.) over the specification of what came to be known as the S-100 bus. Although the subcommittee immediately began working on a number of other proposed standards, microcomputer system buses have remained a major part of its work—unfortunately, with very little practical result. After almost 10 years of effort, only two microcomputer bus standards, S-100 (696) and Multibus (796), have been adopted by the IEEE.

Much else of what was presented as history in the August issue is in conflict with my personal knowledge and the public record. For example, the origins of what became IEEE-STD-802 (Local Networks) and the P896 (originally the Advanced Microcomputer System Backplane Bus) are misrepresented. The following is the text of the first two paragraphs of a report on the status of the P896 activity in the February 1981 issue of *Micro* ("Status Report on the P896 Backplane Bus," Andrew A. Allison, p. 67):

"A subcommittee on microprocessor standards was set up by the IEEE Computer Society in August 1977. By the middle of 1978, the committee's efforts toward developing standard specifications for the S-100 (P696) and Multibus (P796) buses had made clear the need to consider future systems bus requirements before the emergence of yet another generation of de facto but incompletely specified and incompatible buses.

The working group set up to consider this need [This was the Future Bus (*not* Futurebus) subgroup chaired by Cash Olsen.] concluded that the buses then being specified by the Microprocessor Standards Committee could not be extended to satisfy the requirements anticipated for future microprocessor-based systems. Three major categories of bus—backplane, local network, and residential—were identified. A backplane bus subcommittee was set up (by the present writer) in June 1979, and Project Authorization Request Number 896 was approved by the IEEE Standards Board in September of the same year. EDSIG—the European Distributed Intelligence Study Group—set up a subgroup in May 1980 to interact with the 1EEE work. EDS1G is one of the working groups supported by the Commission of European Communities for promoting standardization in the field of data processing."

This makes it clear where the P896 and 802 activities originated. The fact that Maris Graube quickly took the local network effort out from under the MSC's purview is no doubt the reason that it is now an IEEE standard.

The status report was based on the working document for the Boulder P896 workshop and included specification of the serial link feature allegedly introduced by me at the workshop. The position attributed to me in the August issue is, quite simply, false. Credit for the development of this feature, part of P896 from its early days and since incorporated into several other buses, and probably the most useful outcome of the P896 effort, belongs (as I informed Stewart when he was researching his paper) to Rollie Linser.

The reference to Versabus in the August article is incorrect. Both Versabus and Nubus (then still in the hands of M.I.T.) were among the preexisting specifications presented to the P896 working group as candidates for standardization, but neither were felt to meet the processor-, manufacture-, and technology-independence objectives set for P896.

Similarly, the decision to present a proposed draft to the MSC for approval to distribute for public comment was the result of a vote of the working group. 1t is ironic that the MSC's January decision to deny that request on the basis of a minority viewpoint has resulted in Versabus's successor, the VMEbus, becoming the de facto standard 32-bit bus. The characterization of that vote in the article in question is, incidentally, not factual—among other things, Nicoud was not even present!

The fundamental reasons for the failure of the MSC to produce useful standards, in my opinion, were (and remain) lack of understanding of the difference between controlled and uncontrolled specifications, and of the needs of the user communities, and the insistence by certain members of the MSC that the proposed standards incorporate their opinions. Perhaps the most ludicrous example of the latter was the holding up of the Multibus draft for months until the working group chairman acceded to Stewart's nomenclature demands.

As noted above, in 1977 the S-100 bus specification was both popular and out of control, with as many implementations as suppliers and serious incompatibilities between them. All of the other preexisting buses taken up by the MSC since that time have been controlled by their proprietors. The failure to recognize this fundamental difference was the cause of the so-called "bus wars," which were (and are) primarily fought over efforts by the MSC to impose, frequently over the objections of the working groups actually drafting the standards, changes to proprietary, de facto standards. The outcome has been that the need which led to formation of the MSC, namely the interoperability of subsystems from different suppliers, has been met by the use of de facto rather than 1EEE standards.

1 submit that the MSC will continue to fail in its obligation to provide useful, timely standards until it recognizes that: (a) microprocessor and computer manufacturers have a (perfectly legitimate) commercial interest in establishing proprietary buses as de facto standards; (b) the MSC has no business ratifying such standards, with or without the cosmetic changes that are the only kind possible for this type of standard; (c) the plethora of overlapping bus specifications being "standardized" defeats the objective of standards development; and (d) the marketplace will continue to establish de facto standards if **adequate** (as opposed to wonderful) alternatives are not offered in timely fashion.

The foregoing is, unfortunately, probably irrelevant to microcomputer system bus standards development. The IBM PC and PC AT buses will clearly remain the de facto standards for 8- and 16-bit subsystems development for the foreseeable future. Absent something dramatic from IBM very soon, VMEbus's present domination of the 32-bit arena will also be secure. In other words, the war is over!

Andrew Allison

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 183 **Medium** 184 **Low** 185

MicroReview

David L. Hannum/AT&T Information Systems

Three for Christmas

Here are three solid products well worth the software buyer's consideration. All three are priced right, and all three have received reasonably good reviews in the computer press.

Mirror. This data communications program is or at least acts like a clone of Crosstalk XVI and therefore could come under fire as a result of recent court rulings. (See MicroLaw, p. 76, for comments on pending court action involving Mirror.—*Ed.*) In the meantime, we should enjoy having a reasonably priced PC software package that not only looks and acts like the de facto leader but improves on it.

Mirror has all the features of Crosstalk plus several Crosstalk does not provide. Mirror can run as a background program, allowing a user to run an application while sending large files to a host. It can answer calls—serve as a bulletin board, for example—while the user is working. It enhances the XMODEM protocol.

As far as I am concerned, the capability to run as a background program and the XMODEM enhancements are the primary reasons one should consider this program.

Mirror runs well, is easy to learn and use (no learning required at all if the user already knows Crosstalk), and handles errors well. Mirror's developers have done a good job with the manual—I found it easy to learn how to do a good VT100 emulation, for example.

I give this product a 9+. It's worth a look—at $49.95, it provides higher performance for price than anything else on the market.

Mirror is available from SoftKlone, 1210 East Park Ave., Tallahassee, FL 32301; (904) 878-8564.

RAM-Resident Printmerge. This is a very specialized piece of software, strictly for HP LaserJet users. It allows the sophisticated user to take advantage of some of the capabilities built into the printer but not available through normal applications. It supports line drawing and boxing of text and allows graphics, charts, and tables to be mixed with text. And since it is a RAM-resident program, it sits in the background until it is needed.

RAM-Resident Printmerge is easy to load, learn, and use, and its manual is adequate for those who already understand both the LaserJet and their PC. It is a "must have" utility for anyone running a PC/LaserJet combo, and at $124 it is not overpriced, given its capabilities. A solid 8, but not for the beginner.

> *Programs that hide in RAM and emerge when needed are quite useful, but they should make us consider what they are doing to the system.*

RAM-Resident Printmerge is available from Polaris Software, PO Box 28789, San Diego, CA 92128; (619) 489-8243.

Referee. Programs that hide in RAM and emerge when needed are quite useful, but they should make us consider what they are doing to the system. Unlike programs in ROM or on disk, they need constant attention or they may seriously degrade the service a PC provides.

RAM-resident programs take computer resources, particularly processor time. This means there may be a fight (collision) between the application a user is running and a RAM-resident program, or even between two or more RAM-resident programs. This possibility calls for the services of a "referee," a program that can tell other programs when to play ball (activate) and when to leave the game (deactivate). One such package, appropriately named Referee, does this efficiently for some but not all RAM-resident packages.

Referee is not a beginner's tool. While it works well with Prokey and Sidekick, it does not function well with certain other programs and may even cause what its user wants to prevent, data losses and system crashes. One must be aware of its quirks. Used knowledgeably, Referee is a good package. Its manual is more than adequate for the type of user who should be running the program. I give Referee a solid 7 with the above reservations.

Referee is available for $69.95 from Persoft Inc., 465 Science Drive, Madison, WI 53711-9380; (608) 273-6000.

Next issue. Look for a review of two new graphics packages for the IBM PC and compatibles—Concorde and Picture Perfect. With the next issue, my term as MicroReview editor expires. I have enjoyed evaluating micro products, services, and books with you and receiving your cards and letters. I welcome my successors, editorial board members Richard Mateosian and Kenneth Majithia. I am sure they will value your suggestions and comments, as I did.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 171 Medium 172 Low 173

New Products

Editor: Marlin H. Mickle/University of Pittsburgh

Send announcements of new microcomputer/microprocessor products, and products for review, to Managing Editor, IEEE MICRO, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

VRTX, Unix optional on 32-bit board

Microbar Systems is shipping its MT68020 single-board computer designed for use in multitasking applications. The MT68020 features a 68020 32-bit processor, the Multibus II open-system architecture, and options in the operating system, DMA, memory management unit, and floating-point coprocessor.

The board is available in 12.5- and 16.67-MHz versions. On-board, 4M-byte, dual-ported dynamic RAM can be increased to 16M bytes with one wait-state with an expansion board. Memory is accessed either by the on-board MPU or by another Multibus II requester over the Parallel System Bus. The message-passing coprocessor supports the Multibus II PSB interface, and a dual-port circuit arbitrates access to RAM.

MT68020 implements two optional operating systems: the AT&T Unix System V, Release 2.0, or the Hunter & Ready VRTX real-time operating system. PROM-resident software supports the initial download of a VRTX application from the Unix host to a VRTX processor and allows it to begin operation.

In addition to a standard 8-bit PROM used to initialize the MT68020, perform diagnostics, and set the configuration registers, there are four 28-pin sockets for users' implementations of firmware in PROM. The sockets accommodate sizes from the 2732 (16K bytes) to the 27512 (256K bytes).

OEM-quantity prices are under $2000; engineering samples cost $3490.

Microbar Systems, Inc., 785 Lucerne Drive, Sunnyvale, CA 94086; (408) 720-9300.

Reader Service Number 40

In-circuit emulator connects to PCs

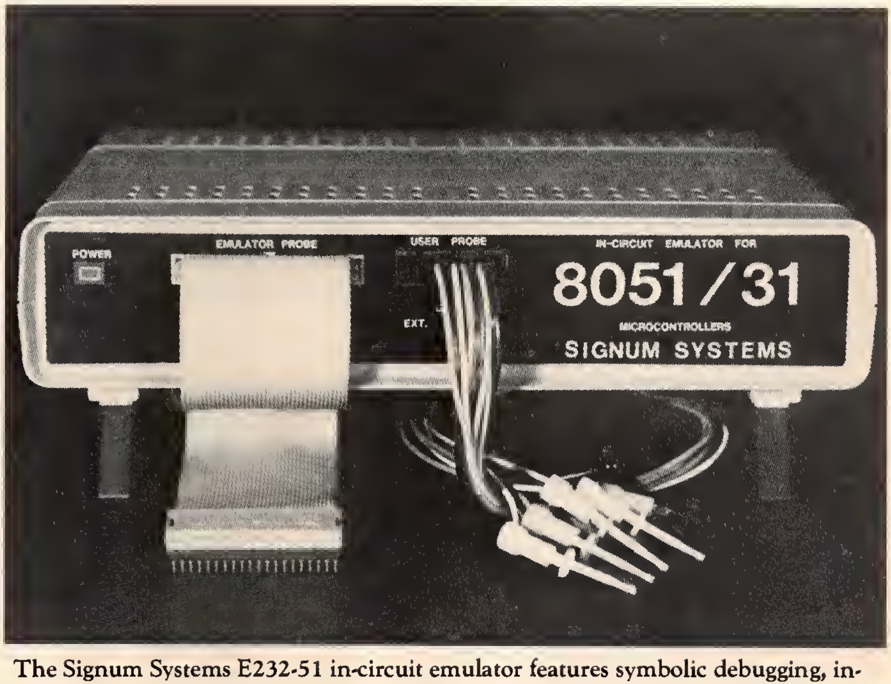
An in-circuit emulator from Signum Systems provides real-time, transparent emulation for the 8031, 8051, and 8751 microcontrollers. Model E232-51, when connected to an IBM PC XT or AT through an RS-232 interface, offers debugging facilities and a user interface with windows, menus, and mouse support.

The real-time trace buffer of the emulator provides information on the address and data buses, status lines, ports 0 through 3, and 11 external user signals. It is capable of stopping the recording process after a specified number of instructions or cycles so the executing program can be recorded without stopping the microcontroller.

Model E232-51 with 64K bytes of overlay program memory is priced at $3195. Optional 8051 relocatable cross-assembler and mouse are available for $199.50 and $99.

Signum Systems, 1820 14th Street, Suite 203, Santa Monica, CA 90404; (213) 450-6096.

Reader Service Number 41



The Signum Systems E232-51 in-circuit emulator features symbolic debugging, in-line assembler and disassembler, 128K bytes of address breakpoints, and an 11-channel user logic state analyzer. It requires a terminal for stand-alone operation or an IBM PC or compatible with 128-K RAM, one serial communication port, and a monochrome or graphics adapter card with monitor.

RAM modules support VMEbus and VMXbus

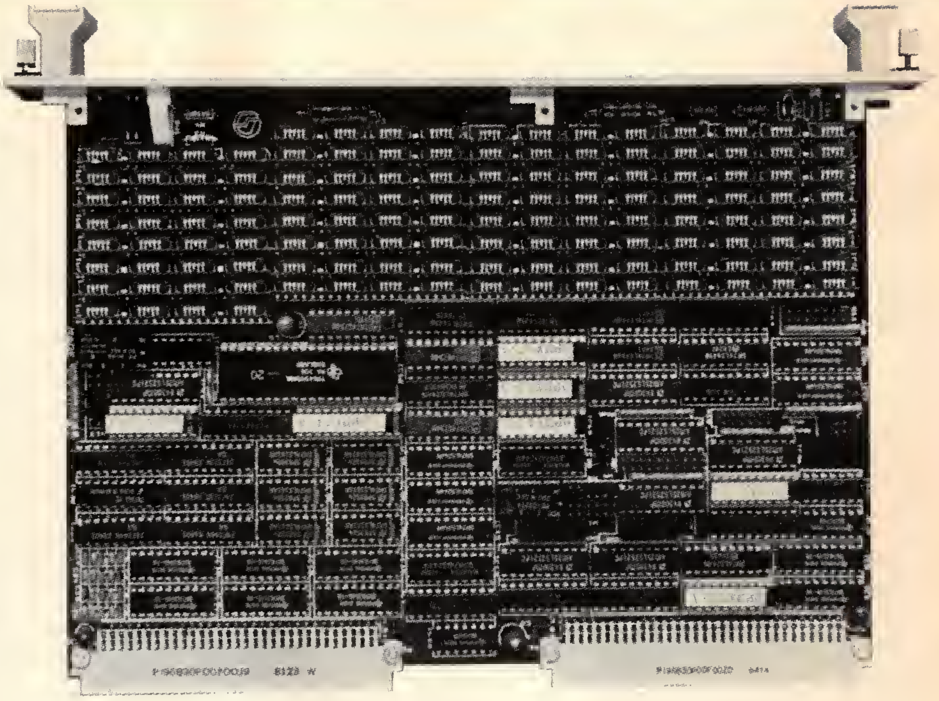
The DSSEDPDX from Data-Sud Systems is a dual-ported RAM module that supports both the VMEbus and the VMXbus, revisions A and B. A -1 version is supplied with 512K bytes of 64K × 4 SIP dynamic RAM, and a -2 version comes with a capacity of 1M byte.

The DSSEDPDX is an expanded bus, double VME board that occupies one slot. Its standard front panel (for double Eurocard cages) incorporates status LEDs for VMEbus/VMXbus access, write-protect switches for both buses, and an AMP connector for broadcast mode. P1 and P2 are both 96-pin DIN41612 connectors.

The DSSEDPDX-1 is priced at $1495; the DSSEDPDX-2 costs $1995. Delivery for the DSSEDPDX boards is from stock to four weeks.

Data-Sud Systems/U.S., Inc., 5025 South Ash Avenue, Bldg. B, Suite 5, Tempe, AZ 85282; (602) 345-0945.

Reader Service Number 42



The Data-Sud Systems DSSEDPDX module is a VMEbus A32 slave capable of driving and monitoring 32 data lines for 32-bit data transfers. The module's VMXbus base memory address is selected by jumper; it decodes 24 address lines and 32 data lines on the VMXbus.

MC68030 production promised for fall 1987

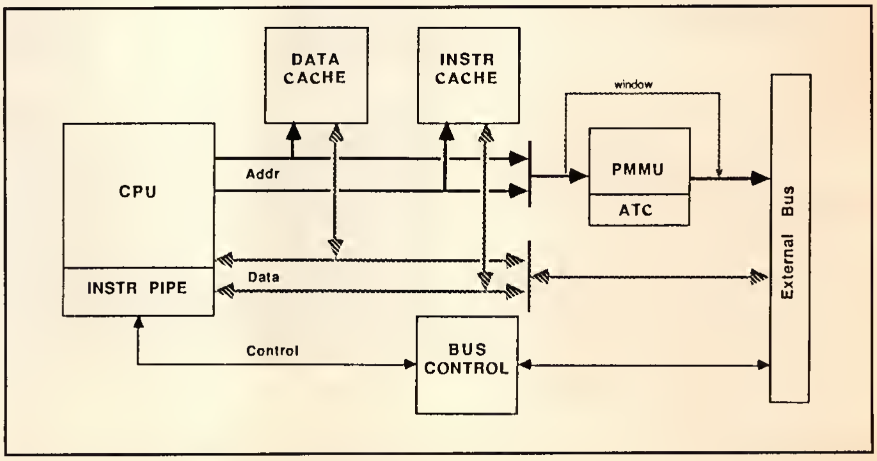
Motorola's Microprocessor Products Group has announced the MC68030, a second-generation 32-bit microprocessor unit. According to the company, the enhanced 16.67-MHz MPU offers twice the performance of its MC68020 and maintains upward software code compatibility with the M68000 family MPUs.

Performance improvements include increased internal parallelism, dual on-chip caches with a burst fillable mode, dual internal data and address buses, improved bus interface, and an on-chip paged memory management unit. The Harvard-style architecture provides the processor with an internal bus bandwidth of more than 80M bytes/s. The on-chip memory management unit reduces the minimum physical bus cycle time to two clocks, one half the time required by the MC68020 and MC688851.

Motorola's high-density MC68030 contains about 300,000 transistors and is sized on a side at about 378 mils. The chip is enclosed in a 128-lead PGA package. Sampling is planned for July, production for October, and a VMEbus-microcomputer version for fourth quarter 1987. Pricing was not released.

Motorola Inc., Microprocessor Products Group, PO Box 3600, Austin, TX 78764; (512) 440-2839.

Reader Service Number 43



Block diagram of Motorola's MC68030 MPU.

Second-generation 32-bit FPC from Motorola

Motorola Microprocessor Products Group has announced a second-generation, 32-bit floating-point coprocessor, which is expected to offer two to four times the performance of the MC68881. The MC68882 enhanced FPC conforms to IEEE 754, the standard for binary floating-point arithmetic. It offers add, subtract, multiply, divide, and transcendental and non-transcendental functions.

The HCMOS VLSI device is designed to operate primarily as a coprocessor with the MC68020 and MC68030 32-bit MPUs through a transparent MC68000 coprocessor interface. In addition, the FPC can be used with M68000-family MPU devices and as a peripheral to non-M68000 processors.

The 16.67-MHz MC68882 FPC is enclosed in a 68-lead PGA package; it is expected to be available for sampling in April 1987 with production planned for August.

Motorola, Inc., Microprocessor Products Group, PO Box 3600, Austin, TX 78764; (512) 440-2839.

Reader Service Number 44

Intel boards combine Multibus, 80386 features

Four single-board computers from Intel Corporation use a 16-MHz, 80386 32-bit microprocessor and a dual-bus structure to provide high-end processing power for intricate applications. The iSBC 386/21, /22, /24, and /28 computers are supported by iRMS 286, Xenix, Unix System V, and any proprietary operating system written for the 8086 or 80286 CPU.

The boards provide up to 8M bytes of 32-bit memory, which can be expanded to 16M bytes with add-on surface-mount modules. The increased memory provides users with direct CPU access to memory through a 64K-byte zero-wait-state cache memory without having to go out over the system bus.

List prices are $4800 for the 386/21, $5970 for the /22, $8310 for the /24, and $12,990 for the /28.

Intel Corporation, 3065 Bowers Avenue, PO Box 58065, Santa Clara, CA 95052-8065; (503) 640-7399.

Reader Service Number 45

Three buses speed CMOS 32-bit multiplier

Advanced Micro Devices' Am29C323 is a 125-ns CMOS 32 × 32-bit parallel multiplier. The first member in a planned CMOS family of 32-bit microprogrammable building blocks, the Am29C323 uses less than one watt of power while operating at 8MHz.

The device's three buses contain two 32-bit input and one 32-bit output ports. It provides individual register feedthrough controls, byte-parity checking on both input ports, and parity generation on the output port. Dual-precision registers on each data input port support multiprecision multiplication. A 64-bit product and a 3-bit overflow product permit the accumulation of values larger than the normal accumulator width.

During 1987 the company expects to introduce additional family products such as a 32-bit floating-point processor, 16-bit microprogram sequencer, 32-bit extended-function ALU, and 64 × 18 dual-access register file. The 125-ns Am29C323 is in production now; 100-ns and 80-ns versions are planned.

The 168-pin PGA-packaged multiplier costs $245 in quantities of 100.

Advanced Micro Devices, 901 Thompson Place, PO Box 3453, Sunnyvale, CA 94088; (408) 982-7448.

Reader Service Number 46

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 192 Medium 193 Low 194

Advertisers

Products

Coming in *IEEE Micro*

February: More on operating systems, multiprocessing, and digital signal processors

February's articles supplement our August, October, and December special issues. Included are an examination of performance models for Unix-based network file systems, a description of the Heidelberg Polyp system—a fault-tolerant multi-microprocessor, and an analysis of alternatives for implementing the discrete Fourier transform in signal processing applications.

FOR DISPLAY ADVERTISING INFORMATION CONTACT

Southern California and Mountain States: Richard C. Faust Company, 24050 Madison Street, Suite 100, Torrance, CA 90505; (213) 373-9604.
Northern California and Pacific Northwest: Don Farris Company, 161 W. 25th Ave., #102B, San Mateo, CA 94403; (415) 349-2222.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070; (408) 741-0354.
East Coast: Hart Associates, Inc., P.O. Box 339, 42 Lake Blvd., Matawan, NJ 07747; (201) 583-8500.
New England: Arpin Associates, P.O. Box 227, 51 Colchester, Weston, MA 02193; (617) 899-5613.
George Watts, III, 4 Conifer Dr., Wilbraham, MA 01095; (413) 596-4747.
Midwest: Thomas Knorr, Knorr MicroMedia, Inc. 333 North Michigan Ave. Chicago, IL 60601; (312) 726-2633.
Southwest: The House Company, 3000 Weslayan, Suite 345, Houston, TX 77027; (713) 622-2868.

Advertising Director: Mike Koehler, *IEEE MICRO*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-3240, 821-8380.

For production information, conference or classified advertising contact Carole L. Porter, *IEEE MICRO*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, (714) 821-1140.

IEEE MICRO

For further information on advertised products, new products, or literature, fill out the **Reader Service Card** (top). Circle the number on the RS Card that corresponds to the number of the item for which you would like more information.

To indicate your interest in an article or department, fill out the **Reader Interest Card** (bottom). Circle the number on the RI Card that corresponds to the level of interest given in the Reader Interest Survey at the end of the article or department.

Please print or type your name and address.

READER SERVICE CARD

IEEE MICRO INFORMATION ABOUT PRODUCTS

Void after June 30, 1987 **12/86**

Name ______________________

Company ______________________

Address ______________________

City ______________________

State ______________________

Zip ______________________

Country ______________________

Title ______________________

Telephone number (   ) ______________________

PRODUCTS PURCHASED OR SPECIFIED	FOR JOB	FOR HOBBY
Computers	80	81
Peripherals	82	83
Data communications equip.	84	85
Memories, components	86	87
Software and services	88	89
Publications	90	91
Other	92	93

94 Please send me information on advertising in *IEEE Micro*.

95 Please send me the IEEE-CS *Publications Catalog*.

96 Please send me an IEEE Fellow nomination form.

97 Please send me an IEEE Computer Society membership application.

98 Please send me an *IEEE Micro* author's guide.

Product announcements, and products for review, should be sent to Marie English, Managing Editor, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

Send more information on numbered items:

1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	97
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94	98
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91	95	99
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100

READER INTEREST CARD

IEEE MICRO EDITORIAL RESPONSE

Void after June 30, 1987 **12/86**

Name ______________________

Company ______________________

Address ______________________

City ______________________

State ______________________

Zip ______________________

Country ______________________

Title ______________________

Telephone number (   ) ______________________

Reader Interest Survey:

101	116	131	146	161	176	191
102	117	132	147	162	177	192
103	118	133	148	163	178	193
104	119	134	149	164	179	194
105	120	135	150	165	180	195
106	121	136	151	166	181	196
107	122	137	152	167	182	197
108	123	138	153	168	183	198
109	124	139	154	169	184	199
110	125	140	155	170	185	200
111	126	141	156	171	186	201
112	127	142	157	172	187	202
113	128	143	158	173	188	203
114	129	144	159	174	189	204
115	130	145	160	175	190	205

Comments:

Do you like *IEEE MICRO's* 1987 editorial calendar (see inside back cover)? What specific articles on the topics listed would you like to see?

Continue comments on other side

This PO box for reader
service cards only.

PLACE
STAMP
HERE

IEEE MICRO

Reader Service Inquiries
Box 24168
Los Angeles, CA 90024
USA

Comments on articles and other editorial matter:

I liked ____________________

____

____

____

____

I disliked ____________________

____

____

I would like ____________________

____

____

____

For Reader Interest Survey, see other side

PLACE
STAMP
HERE

IEEE MICRO

10662 Los Vaqueros Circle
Los Alamitos, CA 90720-2578
USA

1987 EDITORIAL CALENDAR

IEEE Micro—a bimonthly publication of the Computer Society of the IEEE—focuses on helping the designers and users of microprocessor and microcomputer systems explore, produce, evaluate, and apply the latest technologies so that business and research objectives can be achieved.

Feature articles in *IEEE Micro* are original works relating to the design, performance, or application of microprocessors and microcomputers. Tutorial material, industry views, and discussions of standards are often selected for publication. All manuscripts are subject to a peer-review process consistent with most professional-level technical publications. This review may take up to four months.

AD CLOSING DATE: 1st of month preceding issue (Jan. 1st for February issue)

FEBRUARY

DIGITAL SIGNAL PROCESSING, OPERATING SYSTEMS, MULTIPROCESSING

Additional articles supplementing the August, October, and December 1986 issues. Titles include the 80386 plus Unix, performance analysis of Unix-based network file systems, DFT implementations, and the Heidelberg Polyprocessor system.

APRIL

JAPANESE SPECIAL ISSUE: TRON 32-BIT MICROPROCESSORS

IEEE Micro editorial board member and TRON architect/systems designer Ken Sakamura from the University of Tokyo offers a fine collection of articles about Japan's newest offering, The Realtime Operating System Nucleus.

JUNE

NEW DEVELOPMENTS IN MICROPROCESSORS

Explore the latest in design technologies and applications with this issue. Coverage of the Fairchild Clipper, the Intel 80387 coprocessor, and other innovative chips is planned.

Deadline for articles: January 1, 1987

AUGUST

SYMBOLIC PROCESSORS AND SYSTEMS

Articles will discuss the architecture, performance, and application features of specialized microprocessors that incorporate AI languages in hardware.

Deadline for articles: March 1, 1987

OCTOBER

EUROPEAN SPECIAL ISSUE

Catch up with the industry's newest technologies from Europe. Guest editor is Karl E. Grosspietsch, scientist at the German national research institute for mathematics and data processing, the Gesellschaft fuer Mathematik und Datenverarbeitung, in St. Augustin, West Germany.

Deadline for articles: April 1, 1987

DECEMBER

THE NEW TECHNOLOGIES

Read the latest information concerning subjects such as GaAs and one-micrometer technologies and high degrees of silicon integration.

Deadline for articles: June 1, 1987

Articles may change. Please contact the editors to confirm.

HOW TO SUBMIT AN ARTICLE TO IEEE MICRO

Prospective contributors should submit their manuscripts directly to:

James J. Farrell III
Editor-in-Chief, *IEEE Micro*
VLSI Technology Incorporated
10220 South 51st Street
Phoenix, AZ 85044
(602) 893-8574

Successful contributions will be original works with sufficient introductory material and at least 20 percent of the total length devoted to tutorial material. The tutorial section will describe the principles or techniques of existing approaches and evaluate their advantages and disadvantages. Furthermore, the contributions will describe the practical or potential applications of the material presented. To improve readability, the discussion will be augmented with examples, tables, diagrams, charts, and photographs.